VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PLÁNOVÁNÍ CESTY POMOCÍ VORONOIOVÝCH DIAGRAMŮ

PATH PLANNING USING VORONOI GRAPHS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADAM ŽIVČÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAROSLAV ROZMAN, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/18391/2017/xzivca03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Živčák Adam**
Obor: Informační technologie
Téma: **Plánování cesty pomocí Voronoiových diagramů**
Path Planning using Voronoi Graphs
Kategorie: Umělá inteligence

Pokyny:

1. Nastudujte Robotický operační systém (ROS) a metody pro plánování cesty, především ty, založené na Voronoiových grafech.
2. Navrhněte zásuvný modul založený na Voronoiových grafech pro plánování cesty do ROS.
3. Navržený zásuvný modul implementujte ve zvoleném programovacím jazyce (C++ nebo Python).
4. Ověřte kvalitu plánování cesty a srovnejte ji se stávajícími implementovanými algoritmy.

Literatura:

- Howie Choset et al., Principles of Robot Motion, 2005, ISN 0-262-03327-5.
- Robotický operační systém, www.ros.org.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rozman Jaroslav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Náplňou tejto bakalárskej práce je návrh a implementácia zásuvného modulu pre plánovanie cesty do robotického operačného systému s využitím Voronoiových diagramov. Plánovanie cesty prebieha v prostredí ktoré je pre robota známe, teda pozná jeho topológiu a rozmiestnenie prekážok. Práca obsahuje prehľad základných súčasti robotického operačného systému, vysvetlenie Voronoiových diagramov a algoritmov pre ich zostrojenie. V poslednej časti sa nachádza porovnanie implementovaného modulu s plánovačmi integrovanými v ROS.

Abstract

The main purpose of this bachelor thesis is to design and implement plugin for robot operating system, which will be used for path planning using Voronoi graphs. Path planning is executed in well known world, about which robot knows where obstacles are placed. Thesis contains overview of main concepts of robot operating system, description of Voronoi graphs and algorithms to construct them. In conslusion is placed comparison of implemented plugin for path planning with ROS integrated planners.

Kľúčové slová

robotický operačný systém, ROS, zásuvný modul, navigácia, plánovanie cesty, Voronoiové diagramy, RViz

Keywords

robot operating system, ROS, plugin, navigation, path planning, Voronoi graph, RViz

Citácia

ŽIVČÁK, Adam. *Plánování cesty pomocí Voronoiových diagramů*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Plánování cesty pomocí Voronoiových diagramů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jaroslava Rozmana, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Adam Živčák
17. mája 2018

Poďakovanie

Ďakujem vedúcemu práce, pánovi Ing. Jaroslavovi Rozmanovi, Ph.D. za poskytnuté rady a ústretový prístup pri písaní práce.

Obsah

Kapitola 1

Úvod

Vývoj a využívanie robotických systémov v dnešnej dobe veľmi rýchlo napreduje a rozmáha sa, aj napriek obtiažnosti pri ich vývoji. Avšak v poslednom čase, má možnosť vďaka robotickému operačnému systému (ROS), zapojiť sa do vývoja takmer každý. ROS poskytuje rôzne prostriedky pre prácu s robotmi, od navigácie, lokalizáciu, mapovanie prostredia, či ovládanie jednotlivých ramien robota.

Práve navigácia robota je považovaná za kľúčovú časť autonómnych systémov, ktoré sa využívajú čoraz častejšie v rôznych odvetviach. Pre správnu navigáciu je potrebné aby robot poznal prostredie v ktorom sa pohybuje, prípadne aby bol schopný si ho na začiatku zmapovať. Táto práca sa zaoberá navigáciou, plánovaním cesty, v priestore, s ktorým je robot oboznámený, teda vie, kde sa nachádzajú prekážky a kadiaľ môže viesť cesta.

Vyhýbanie sa prekážkam v dostatočnej vzdialenosti je dôležité, aby nedochádzalo ku kolíziám a robot sa bezpečne dostal na miesto určenia. V tejto práci sa pre nájdenie cesty, z počiatočnej do cieľovej pozície robota, využívajú hrany Voronoiového diagramu, ktoré vedú stredom medzi dvoma prekážkami a tým sa minimalizuje riziko zrážky s prekážkou.

V jednotlivých kapitolách práce si predstavíme robotický operačný systém a jeho základné súčasti. Ďalej sa pozrieme na Voronoiové diagramy, ich vlastnosti a na možnosti ich zostrojenia. Následne navrhneme algoritmy ktoré sa použijú v implementácii a v závere porovnáme novo-implementovaný plánovač cesty s tými, ktoré sú implementované v robotickom operačnom systéme.

Kapitola 2

ROS

ROS (*Robot Operating System*) je meta-operačný systém ktorý sa používa pri návrhu a vývoji rôznych aplikácii pre roboty. Poskytuje služby ktoré očakávame od bežného operačného systému, vrátane hardvérovej abstrakcie, nízko-úrovňového riadenia, zasielania riadiacich správ medzi procesmi a správy ďalších komponent [29]. Taktiež poskytuje nástroje a knižnice pre získavanie, tvorbu a beh programov na rôznych platformách. ROS je licencovaný BSD (*Berkeley Software Distribution*) licenciou, čo je jedna z najslobodnejších open-source licencii.

ROS má veľmi veľkú podporu zo strany používateľov. Na jeho vývoji sa aktuálne (november 2017) podieľa 107 škôl, 43 spoločností, 24 výskumných inštitútov a niekoľko ďalších skupín ľudí z celého sveta [16]. Taktiež existuje fórum, kde je možné zadávať otázky, na ktoré ostatní používatelia odpovedajú, alebo prípadne hľadať odpovede v už zodpovedaných otázkach. Na tomto fóre je registrovaných viac ako 18000 používateľov [6].

2.1 Koncepcia

Celý ROS sa člení na tri úrovne, ktorými sú: *súborový systém*, *výpočtový graf* a *komunita* [18]. Súborový systém tvoria dáta, ktoré môžeme fyzicky ukladať na disk a modifikovať ich. Patria tu balíčky (*packages*) a meta-balíčky (*meta-packages*), ďalej súbory ktoré špecifikujú jednotlivé balíčky (*Manifest*), a taktiež typy správ (*msg*) a služieb (*srv*).

Výpočtový graf je založený na peer-to-peer sieti a popisuje spôsob komunikácie procesov a spracovanie dát. Medzi jeho komponenty patria: uzly (nodes), Master, server parametrov (*parameter server*), správy (*messages*), témy (*topics*), služby (*services*), a formát pre ukladanie a opätovné používanie dát zo správ (*bags*).

Komunita je tvorená distribúciami, skladom dokumentov (*repository*), vlastnou kolekciou Wiki stránok a fórom pre otázky od používateľov a blogom.

2.1.1 ROS distribúcie

Distribúcia robotického operačného systému je súhrn balíčkov určitej verzie, ktorá je pravidelne vydávaná. Ich cieľom je, aby mohli používatelia a vývojári pracovať s aktuálnymi a maximálne stabilnými nástrojmi.

Existuje niekoľko pravidiel pre vydávanie distribúcii. Nová distribúcia ROS vychádza každý rok v máji. Verzia vydaná v nepárnom roku je typu LTS (*Long Term Support*), čiže dlhodobo podporovaná. V párnom roku vychádza normálna verzia s dvoj-ročnou podporou. Keďže ROS beží primárne nad operačným systémom Ubuntu, existuje závislosť medzi ich

verziami. Jednou z nich je že každá ROS distribúcia je podporovaná len na jednej LTS Ubuntu verzii a žiadne dve ROS distribúcie nepoužívajú rovnakú Ubuntu verziu [19].

Od roku 2010 vzniklo jedenásť verzii. Ich názvy sú abecedne zoradené, pričom prvý z nich je *Box Turtle*. Ďalšie sú *C Turtle, Diamondback, Electric, Fuerte, Groovy, Hydro, Indigo, Jade, Kinetic* a *Lunar*. Aktuálne podporované sú Indigo (do apríla 2019), posledná dlhodobo podporovaná verzia Kinetic (do apríla 2021) a posledná vydaná verzia Lunar (do mája 2019).

2.1.2 ROS 2.0

Myšlienka robotického operačného systému a jeho vývoj začal v roku 2007, odkedy sa toho v robotike ale aj v komunite okolo tohto operačného systému veľa zmenilo. Preto začala skupina ľudí pracovať na novej verzii ROS 2.0. Cieľom tohto projektu je vyťažiť z pôvodného ROS to, čo je v ňom dobré a vylepšiť to, čo nie je.

Vývoj ROS 2 začal niekedy v roku 2015 a prvá beta verzia vyšla v Decembri 2016. Po nej vyšli ešte ďalšie dve beta verzie a to v júli a septembri 2017. Prvá (non-beta) verzia vyšla 8. decembra 2017 a volá sa *Ardent Apalone*. Je podporovaná na operačných systémoch Ubuntu 16.04, Mac OS X 10.12 a aj na Windows 10 [5]. Nové verzie verzie by mali vychádzať dva-krát častejšie ako u ROS 1, teda raz za šesť mesiacov.

ROS 2 prináša oproti pôvodnému systému mnoho zmien, pričom my si spomenieme aspoň niektoré z nich. Prvou, z pohľadu iných operačných systémov dosť zásadnou, zmenou je to, že ROS 2 je priebežne testovaný nie len na Ubuntu, ale aj na operačnom systéme OS X El Capitan, a taktiež na Windows 10. Zmeny nastali aj v používaných programovacích jazykoch. Nový ROS sa zameriava na použitie štandardu C++11 a niektorých časti C++14, pričom v ROSe 1 sa používal štandard C++03. Rovnako aj jazyk Python vyžaduje novšiu verziu 3.5, kým ROS 1 podporoval Python 2.

Ďalšie zmeny sa týkajú používania menných priestorov (*namespaces*), používania správ a služieb s rovnakými názvami, ďalšie zavádzajú zjednotenie časových typov a mnoho ďalších úprav či vylepšení [3]. Zmeny ktoré prichádzajú s verziou ROS 2 by v princípe bolo možné začleniť do existujúceho systému ROS. Avšak ľudia ktorí sú za vývojom ROS 2 sa zhodli na tom, že ak chcú dosiahnuť všetky výhody ktoré má ROS 2 priniesť, tak by bolo veľkým rizikom meniť aktuálny systém, na ktorý sa spolieha veľké množstvo ľudí. Preto bude ROS 2 vyvíjaný ako paralelná množina balíčkov, ktorá bude schopná spolupracovať s ROS 1 pomocou tzv. *message bridges*. Pritom pôvodný robotický operačný systém bude naďalej existovať a nebude ovplyvňovaný vývojom ROS 2.

2.1.3 Prekladový Systém

Prekladový systém (*Build system*) je zodpovedný za vytváranie cieľov, ktoré budú následne používateľmi spúšťané. Tieto ciele sa generujú počas prekladu zdrojového kódu a ich forma môže byť rôzna. Sú to napríklad knižnice, spustiteľné programy, vygenerované skripty, rozhrania a mnohé iné [1]. Medzi všeobecne známe prekladové systémy patria *GNU Make*, *CMake*, či *Apache Ant*. Pre ich beh sú potrebné konfiguračné súbory, v ktorých sú informácie o mieste uloženia zdrojového kódu a jeho závislostiach, o tom aké ciele majú byť vytvorené alebo kde majú byť uložené.

V robotickom operačnom systéme existujú dva prekladové systémy. Prvý z nich je pôvodný *rosbuild*, a jeho nástupca *catkin*. Rosbuild je staršia verzia, ktorú sa už v nových systémoch neodporúča používať, keďže má niekoľko nevýhod, ktoré sa v novšom catkin systéme nenachádzajú. Hlavnou z nich je prenosnosť medzi viacerými operačnými systémami.

Rosbuild počas prekladu kódu spúšťa niekoľko skriptov, používa GNU Make a taktiež aj CMake. Catkin je z tohto hľadiska jednoduchší a vystačí si s CMake, čo nie je problém použiť ani v operačnom systéme Microsoft Windows. Jeho postup prekladu je veľmi podobný ako v CMake, ale naviac umožňuje automatické vyhľadávanie balíčkov a pridáva možnosť prekladať súčasne viacero navzájom závislých projektov. V tomto dokumente budeme uvažovať a pracovať už len so systémom catkin a spôsobmi jemu prispôsobenými.

2.1.4 Workspace

Balíčky v robotickom operačnom systéme môžu byť vytvárané a prekladané ako nezávisle projekty po jednom, alebo ako skupina niekoľkých naraz. K tomu aby sme to mohli robiť druhým z uvedených spôsobov sa používa technika pracovných prostredí.

Pracovné prostredie (ang. *workspace*) je priečinok, kde sa ukladajú, modifikujú, prekladajú a inštalujú balíčky [2]. Môže obsahovať až štyri rôzne pod-priestory, pričom každý z nich plní v procese vývoja softvéru špecifickú úlohu. Medzi tieto pod-priestory patria:

- `src/` - miesto pre zdrojový kód
- `build/` - miesto pre spustenie prekladu
- `devel/` - miesto na uloženie preložených cieľov
- `install/` - inštalácia preložených cieľov

2.1.5 Packages

Balíčky (ang. *packages*) sú hlavnou jednotkou organizácie softvéru v ROS. Môžu obsahovať uzly, knižnice, konfiguračné súbory a ďalšie časti ktoré vytvárajú moduly. Ich hlavnou prioritou je aby bol kód znovu-použiteľný a aby jeho správa bola jednoduchšia. Je to najatomickejšia časť, ktorú je možné v ROS preložiť. Každý balíček tvorí priečinok, ktorý má predpísanú štruktúru a musí obsahovať minimálne súbor `package.xml` [10]. Štruktúra tohto priečinka je nasledujúca:

- `include/` - priečinok s hlavičkovými súbormi
- `msg/` - priečinok obsahujúci typy správ
- `src/` - priečinok obsahujúci zdrojové súbory
- `scripts/` - priečinok so spustiteľnými skriptami
- `CMakeLists.txt` - CMake súbor pre preklad
- `package.xml` - súbor popisujúci balíček

Balíček je možné vytvoriť manuálne, alebo pomocou nástroja `catkin_create_pkg`. Syntax tohto príkazu je:

```
$ catkin_create_pkg nazov_balicka [zavislost_1] [zavislost_2]
```

kde sú závislosti voliteľnými parametrami a sú oddelené medzerami. Odporúčanou technikou, je vytvoriť si pracovné prostredie (*workspace*) a následne v ňom vytvárať balíčky. Ak máme toto prostredie vytvorené, tak sa presunieme do priečinka `src` v ňom a následne pomocou príkazu `catkin_create_pkg` vytvoríme balíček s názvom *my_package*, ktorý bude závisieť na balíčkoch *std_msgs*, *rospy* a *roscpp*. Príkald:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg my_package_name std_msgs rospy roscpp
```

To nám vytvorí priečinok `my_package` a v ňom súbory `package.xml` a `CMakelists.txt`, ktoré už budú naplnené základnými informáciami o našom balíčku [4]. Taktiež bude v týchto súboroch pomocou komentárov naznačené, čo a akým spôsobom je ešte možné doplniť.

Preklad balíčka sa spúšťa z koreňového priečinka pracovného adresára, pomocou príkazu `catkin_make`. Po jeho dokončení sa vytvorí hierarchia súborov v zložke `catkin_ws/devel`. Aby sme sa na balíček mohli odkazovať a ďalej ho používať, tak je potrebné ho zaregistrovať. Na to existuje vo vytvorenej zložke skript s názvom `setup.bash`, ktorý je potrebné spustiť. Po tomto kroku môžeme jeho názov používať v nástrojoch ako sú *rospack*, *roscd* a iných.

2.1.6 Manifest

Je to XML súbor nazvaný `package.xml` ktorý je umiestnený v koreňovom adresári balíčka. Obsahuje podrobné informácie o balíčku – názov, číslo verzie, autorov, správcov a závislosti na iných balíčkoch [8]. Ak závislosti v súbore chýbajú, alebo sú zapísané nesprávne tak balíček nebude správne fungovať na iných zariadeniach. Dôvodom je to, že informácie o závislostiach sú potrebné na to, aby si používateľ balíčka mohol zabezpečiť požadované časti softvéru.

2.1.7 Popis správ a služieb

ROS používa súbory s jednoduchou syntaxou pre popis hodnôt, ktoré môžu uzly vysielať. Tieto súbory majú príponu `.msg`, a sú uložené v priečinku `msg/` daného balíčka. Popis v nich sa následne využíva pre automatické generovanie zdrojového kódu správ v jednom z podporovaných jazykov [9]. Tieto súbory môžu mať dve časti. Prvou z nich sú polia (ang. *fields*) obsahujúce dáta, ktoré sú posielané správami. Druhou sú konštanty (ang. *constants*) definujúce užitočné hodnoty, ktoré môžu byť použité v poliach.

Pre popis typov ROS služieb sa taktiež používa jednoduchá forma zápisu, ktorá je založená na msg formáte. Tieto súbory sú nevyhnutné pre komunikáciu typu požiadavka – odpoveď medzi uzlami. Majú príponu `.srv` a sú uložené v priečinku `srv/` daného balíčka. Každý `.srv` súbor je zložený z dvoch častí (požiadavka a odpoveď), ktoré majú formát typu `.msg` a sú oddelené pomocou '`---`'. Jednoduchý príklad:

```
string str
---
string str
```

2.1.8 Nodes

ROS je navrhnutý tak, aby bol modulárny, k čomu sa využívajú aj uzly. Uzol je proces ktorý vykonáva nejaký výpočet. Môže to byť lokalizácia robota, spracovanie dát z laserových senzorov, navigácia, kontrola motorov alebo mnohé ďalšie. Niektoré uzly sú prepojené do grafu a navzájom komunikujú pomocou tém, služieb a serveru parametrov. Každý uzol má názov, ktorý ho jednoznačne identifikuje vrámci systému. Taktiež majú typ, ktorý uľahčuje odkazovanie na tento uzol v systéme. Pre vytvorenie uzlu sa využívajú tzv. klientské knižnice, z ktorých sú najpoužívanejšie *roscpp* a *rospy*. Využívanie uzlov má niekoľko výhod pre beh celého systému. Prvou z nich je výraznejšia tolerancia chybových stavov, ktoré nastanú a

ich ošetrenie je riešené len na úrovni daného uzla. Nakoľko je celý systém rozdelený do väčšieho počtu uzlov, tak komplexnosť kódu nie je tak veľká, ako keby to bol jeden celok. Zároveň sú skryté implementačné detaily a stačí používať aplikačné rozhranie [12].

Špeciálny uzol *Master* vykonáva registráciu a vyhľadávanie ostatných uzlov. Taktiež vyhľadáva odosielateľov a prijímateľov tém a služieb. Bez neho by uzly neboli schopné nájsť sa a teda ani komunikovať. Master taktiež poskytuje beh parameter serveru. Spúšťa sa väčšinou pomocou príkazu `roscore`, ktorý spustí aj ďalšie nevyhnutné časti systému [11].

Parameter server je súčasťou uzlu Master a umožňuje ukladať dáta podľa kľúča. Je to zdieľaný slovník dostupný prostredníctvom rozhrania siete. Dáta ktoré sú v ňom uložené, sú parametre ktoré tam uzly za behu ukladajú a získavajú. Najlepšie použiteľný je pre statické nebinárne dáta (napr. parametre nastavenia), ktoré sú potom globálne viditeľné. Nie je navrhnutý ako vysoko-výkonný uzol, ale pre svoje účely je postačujúci [13].

2.1.9 Topics

Témy (ang. *topics*) sú pomenované zbernice, prostredníctvom ktorých sú vymieňané správy. Vo väčšine prípadov, uzly nevedia s kým komunikujú, len odoberajú dáta od nejakej témy, do ktorej iné uzly prispievajú. Témy sú určené pre jednosmernú komunikáciu. Ak nejaký uzol potrebuje komunikovať typom požiadavka – odpoveď môže na to využiť služby.

Každá téma je typovaná typom správy, ktorá môže byť na danú tému vysielaná. Uzly potom môžu odoberať správy so zhodným typom. ROS aktuálne podporuje prenos správ založený na TCP/IP aj na UDP. TCP prenos je známy aj ako *TCPROS*, a vysiela dáta prostredníctvom TCP/IP spojenia. Je to predvolený spôsob komunikácie v ROS a jediný, ktorý je vyžadovaný a každá klientská knižnica ho musí podporovať. Prenos nad UDP sa označuje ako *UDPROS*, a momentálne je podporovaný len v `roscpp` [15]. Tento spôsob prenosu je stratový, ale má nízku latenciu, čo je v niektorých úlohách vítané. Uzly sú schopné oznámiť preferovaný spôsob komunikácie. Ak jeden uzol preferuje UDPROS ale druhý ho nepodporuje, tak komunikácia bude prebiehať pomocou TCPROS.

2.1.10 Services

Predchádzajúci spôsob komunikácie je síce veľmi flexibilný, ale nie je vhodný pre komunikáciu, kedy chceme zasielať požiadavky a prijímať na ne odpovede. Pre tento spôsob sa využívajú služby (ang. *services*), ktoré sú definované párom správ – jedna pre požiadavku a druhá pre odpoveď. Tieto definície sú uložené v `.srv` súboroch, ktoré sa pomocou klientskej knižnice prekladajú na zdrojový kód. Jeden uzol poskytuje službu pod určitým názvom. Druhý (klient) volá túto službu zasielaním požiadaviek a čaká na odpoveď. Klient môže vytvoriť trvalé spojenie so službou čo umožní vyššiu efektivitu na úkor vykonávania zmien u sprostredkovateľa služby.

Podobne ako v témach, aj služby majú definovaný typ. Ten je zložený z názvu balíčka a názvu `.srv` súboru. Služby sú naviac opatrené verziou, ktorá je MD5 sumou `.srv` súboru a uzly môžu volať služby, ak sa táto hodnota a typ služby zhodujú. Tento prístup zaručuje, že kód klienta aj serveru boli preložené zo zhodného základu [14].

2.2 Pluginy v ROS

Zásuvný modul (ang. *plugin*) je softvérová komponenta ktorá rozširuje alebo mení funkčnosť inej komponenty alebo aplikácie. Pluginy majú v ROS veľký význam, nakoľko umožňujú

rozširovať základnú funkcionalitu celého systému. Pre prácu s nimi existuje knižnica *pluginlib*, ktorá slúži na zavádzanie a uvoľňovanie pluginov zo systému. Pluginy v ROS sú dynamické triedy, ktoré sa načítavajú za behu programu z knižnice. Pluginlib ich dokáže načítať z knižnice bez toho aby ju o tom musel upovedomiť. Odpadá teda nutnosť explicitne zlinkovať celú aplikáciu so všetkými komponentami pred jej spustením.

Aby mohla byť trieda s pluginom dynamický načítavaná, musí byť označená ako exportovaná trieda. To sa vykonáva pomocou špeciálneho makra `PLUGIN_EXPORT_CLASS`, ktoré sa vkladá do zdrojového súboru (`*.cpp`) ktorý je súčasťou pluginu [7]. Ďalší súbor, ktorý je potrebný preto, aby mohol systém automaticky detekovať a načítavať pluginy je súbor popisujúci plugin (ang. *plugin description file*). Je to XML súbor, v ktorom sú uložené všetky potrebné informácie o plugine zapísané syntaxou, ktorej porozumie ROS. Obsahuje informácie o knižnici v ktorej sa plugin nachádza, meno a typ pluginu, jeho slovný popis a podobne.

Z dôvodu aby bolo možné nájsť všetky dostupné pluginy v systéme, musí každý balíček explicitne uviesť, aké pluginy exportuje a aké balíčky obsahujú tieto pluginy. Odkaz na pluginy balíčka musia byť uvedené v sekcii `export` v súbore popisujúcom balíček (*package.xml*). Pomocou nástroja `rospack` je možné vypísať všetky pluginy ktoré sú dostupné z daného balíčka. Použitie je následovné:

```
$ rospack plugins --attrib=plugin name_of_package
```

2.3 Navigácia v ROS

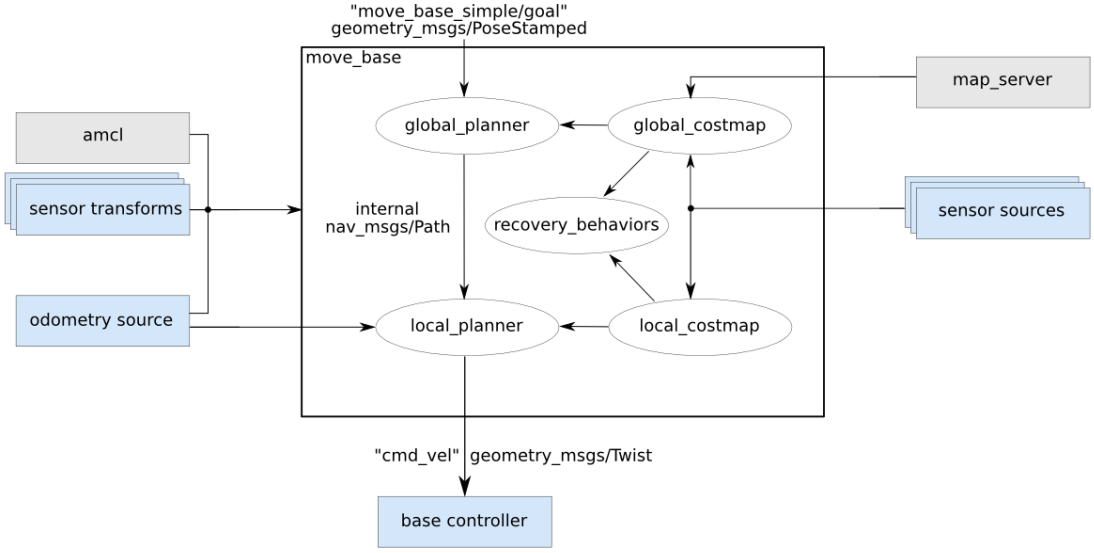
Navigácia je aj v ROS veľmi dôležitou časťou, bez ktorej by bol presun robotov z jedného miesta na druhé veľmi komplikovaný. V robotickom operačnom systéme sa o navigáciu stará tzv. *navigation stack*. Tento navigation stack získava informácie o polohe a rýchlosti robota, a taktiež informácie z máp a na ich základe vytvára a zasiela príkazy pre pohyb robota.

Navigačná vrstva je na konceptuálnej úrovni veľmi jednoduchá. Zbiera informácie od rôznych senzorov a produkuje príkazy pre kontrolu robota, ktoré sa mu zasielajú. Avšak samotné použitie na ľubovoľnom robotovi je o niečo zložitejšie. Základnými prerekvizitami sú aby na robotovi bežal ROS, mal nakonfigurovaný transformačný strom, a publikoval dáta zo senzorov pomocou správnych typov správ [22]. Na obrázku 2.1 je zobrazená štruktúra hlavnej časti navigačnej vrstvy `move_base`. Táto komponenta poskytuje rozhranie pre konfiguráciu, beh a interakciu s navigačnou vrstvou. Časti ktoré sú v obrázku nakreslené ako modré, sa líšia podľa platformy robota. Sivé časti nie sú nevyhnutné pre správne fungovanie, ale sú poskytované pre všetky druhy systémov a biele komponenty sú nevyhnutné a taktiež sú poskytované pre všetky systémy.

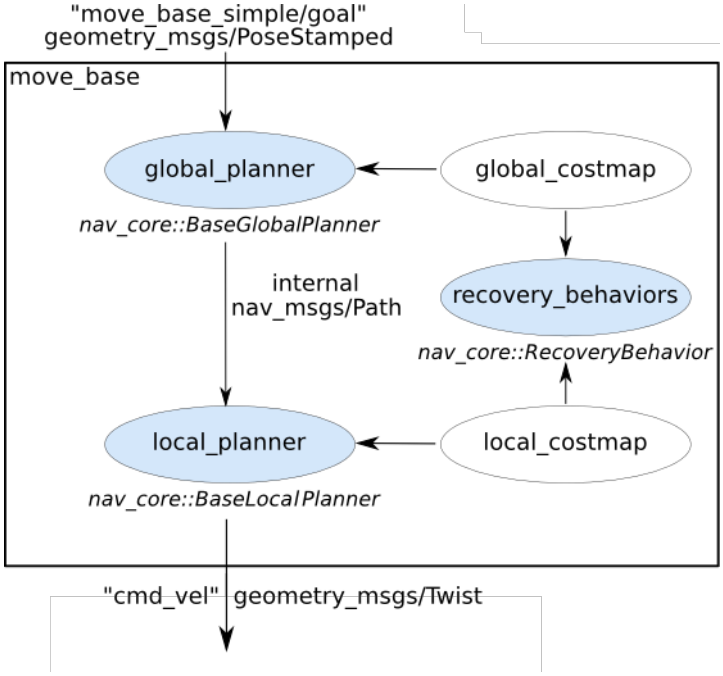
2.3.1 nav_core

Balíček `nav_core` poskytuje rozhrania pre ďalšie časti ROS, ktoré súvisia s navigáciou. Poskytuje rozhranie pre globálny plánovač (BaseGlobalPlanner), lokálny plánovač (BaseLocalPlanner), a pre mechanizmus zotavenia (RecoveryBehavior). Tieto rozhrania umožňujú jednoducho zameniť plánovač či mechanizmus pre zotavenie za nejaký iný [21].

Na obrázku 2.2 sú zobrazené rozhrania balíčka nav_core, ktoré musia byť použité ak chceme použiť nejaký plánovač alebo mechanizmus pre zotavenie.



Obr. 2.1: Štrtruktúra komponenty move_base [20]



Obr. 2.2: Rozhrania navigačnej vrstvy [21]

Rozhranie `nav_core::BaseGlobalPlanner` musí byť použité pre plugin ktorý vykonáva globálne plánovanie, podobne rozhranie `nav_core::BaseLocalPlanner` musí byť použité pre plugin vykonávajúci lokálne plánovanie, a rozhranie `nav_core::RecoveryBehavior` musia použiť pluginy pre mechanizmus zotavenia.

2.3.2 Global planner

Globálny plánovač dostáva informácie o prekážkach a hodnoty z costmapy, informácie z lokalizačného systému robota a cieľovej pozícii v mape. Z týchto informácii vypočíta plán

cesty, ktorej nasledovaním dosiahne robot daný cieľ. Dôležitou vlastnosťou globálneho plánovača je jeho efektivita, aby mohla navigácia bežať v rozumnej rýchlosti a stíhala reagovať na zmeny prostredia.

Medzi tri základné globálne plánovače patria: `global_planner`, `carrot_planner` a `navfn`. `Carrot_planner` je z nich najjednoduchší. Na začiatku overí či zadaný cieľ neleží vo vnútri prekážky. V prípade že áno, tak cieľom určí najbližšie možné miesto k tomuto nedosiahnuteľnému bodu a naplánuje cestu k nemu. V prípade že je cieľ bez problému dosiahnuteľný, tak naplánuje cestu priamo k nemu.

`Navfn` poskytuje rýchlu navigačnú funkciu, ktorá predpokladá použitie okrúhleho robota. Pre nájdenie čo najkratšej cesty z počiatočného bodu do cieľového bodu používa Dijkstrov algoritmus.

`Global_planner` je o niečo zložitejší ako `navfn` a ponúka niekoľko možností nájdenia cesty. Umožňuje použiť Dijkstrov algoritmus, alebo algoritmus A* a nastavenie ďalších parametrov pre prispôsobenie nájdenej cesty.

2.3.3 Local planner

Lokálny plánovač produkuje príkazy pre ovládanie pohybov robota, tak aby sa bezpečne pohyboval smerom k cieľu. Na vstup dostáva plán od globálneho plánovača, ktorý sa snaží nasledovať, ale zároveň berie ohľad na informácie zo senzorov robota, či informácie o prekážkach z costmapy. Medzi najpoužívanejšie lokálne plánovače patria: `base_local_planner`, `eband_local_planner` a `tebd_local_planner`.

Plánovač `base_local_planner` podporuje robotov s akýmkoľvek pôdorysom, ktorý môže byť vyjadrený ako konvexný mnohouholník, alebo kruh. Taktiež podporuje robotov s všesmerovým pohonom, i tie, ktoré vše-smerový pohon nemajú [17]. Základný princíp algoritmu, ktorý využíva techniku známu ako *Dynamic Window Approach (DWA)* je:

1. Diskrétne vzorkovať v priestore robota (dx, dy, dtheta).
2. Pre každú získanú vzorku rýchlosti vykonať simuláciu, a tak predikovať čo by sa stalo ak by sa táto rýchlosť použila v istom časovom úseku.
3. Ohodnotiť každú trajektóriu získanú simuláciou a odstrániť nedovolené z nich (tie ktoré by pretínali prekážky).
4. Vybrať trajektóriu s najlepším ohodnotením a poslať robotovi príkazy pre presun.
5. Odstrániť už nepotrebné výsledky a prejsť znova na prvý bod.

Plánovač `eband_local_planner` poskytuje rozšírenie v tom, že umožňuje produkovať príkazy pre robotov s diferenciálnym pohonom. Ďalší plánovač, `teb_local_planner`, optimalizuje trajektóriu robota s ohľadom na dobu výpočtu a vzdialenosť od prekážok.

2.3.4 Recovery Behavior

Nepríjemnou vecou v navigácii je, že sa robot môže zaseknúť v pozícii, z ktorej sa nedokáže pohnúť ďalej. Existujú preto mechanizmy, ktoré využíva lokálny plánovač, ktoré tento problém riešia. Najznámejšími z nich sú `clear_costmap_recovery` a `rotate_recovery`.

Balíček `clear_costmap_recovery` je jednoduchý mechanizmus pre zotavenie, ktorý vyčistí priestor v mapách obnovením pôvodnej mapy v určitej oblasti okolo robota. Druhý

mechanizmus, `rotate_recovery`, sa snaží priestor vyčistiť otočením robota o 360 stupňov, ak to dovoľujú prekážky v jeho okolí.

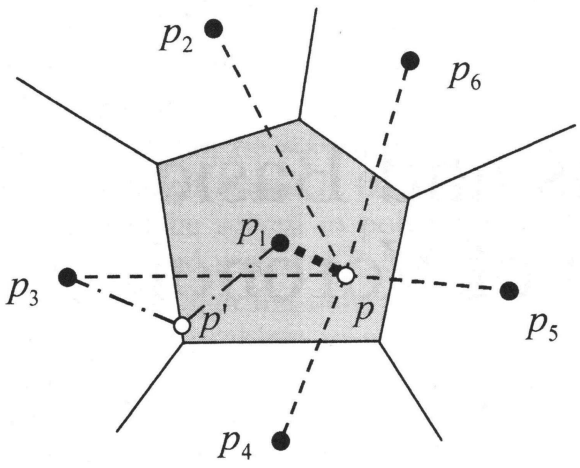
Tieto balíčky spolu s lokálnym plánovačom majú niekoľko parametrov ktoré je možné nastavovať, aby na konkrétnom robotovi fungovali správne. Každý z parametrov má predefinovanú hodnotu, ktorá ale nemusí vyhovovať pre všetky roboty. Jedným z týchto parametrov pre robota Turtlebot je `max_rot_vel`, ktorý udáva maximálnu rýchlosť otáčania robota v radiánoch za sekundu. V základnom nastavení robota Turtlebot je hodnota tohto parametra nastavená na 5.0 a toto nastavenie spôsobuje, že sa robot počas cesty k naplánovanému cieľu zbytočne otáča okolo vlastnej osi. Tento nedostatok je možné čiastočne opraviť tým, že hodnotu parametra `max_rot_vel` zmeníme z 5.0 na 1.0. Zmenu tejto hodnoty hodnoty možno vykonať v súbore `dwa_local_planner_params.yaml`, ktorý sa pre verzziu ROS Kinetic nachádza v priečinku `turtlebot_navigation/param/`.

Kapitola 3

Voronoiové diagramy

Predpokladajme, že máme danú množinu bodov v euklidovskej rovine (body $p_1, \ldots, p_6$ v obrázku 3.1). Počet bodov musí byť konečný, ale väčší ako dva a každé dva body musia byť rôzne, teda také, že sa v rovine nezhodujú. Následne každé miesto v rovine priradíme k bodu, ku ktorému je najbližšie. V prípade že je miesto rovnako vzdialené od dvoch či viacerých bodov v množine, tak priradíme toto miesto ku každému z týchto bodov (bod p' v obrázku 3.1).

Množina miest, ktoré sú priradené k jednému bodu vytvára tzv. región (sivo vyznačená časť v obrázku).Množina miest ktoré sú priradené k dvom alebo viacerým bodom tvorí hranice regiónov (spojité čiary v obrázku). Priľahlé resp. susedné regióny sa zhodujú len v ich hraniciach. Množina regiónov teda spoločne zaberá celý priestor, pričom sú jednotlivé regióny disjunktné (vzájomne sa vylučujúce) okrem ich hraníc ktoré sa prekrývajú. Tieto regióny spolu vytvárajú tzv. *obyčajný rovinný Voronoiov diagram* a jeho jednotlivé časti (regióny) nazývame *obyčajné Voronoiové polygóny* [25].



Obr. 3.1: Obyčajný Voronoiov diagram, prevzaté z [28]

3.1 Definícia

Uvažujme konečný počet bodov n v euklidovskej rovine a predpokladajme že $2 \leq n < \infty$. Body sú označené $p_1, \ldots, p_n$ s kartézskými súradnicami $(x_{11}, x_{12}), (x_{21}, x_{22}), \ldots, (x_{n1}, x_{n2})$ alebo polohovými vektormi $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}$. Body sú rozdielne v zmysle $\mathbf{x}_i \neq \mathbf{x_j}$ pre $i \neq j$, pričom $i, j \in I_n = \{1, 2, \ldots, n\}$. Nech p je ľubovoľný bod v euklidovskej rovine so súradnicami (x_1, x_2), prípadne polohovým vektorom $\mathbf{x}$. Potom euklidovská vzdialenosť medzi bodmi p a p_i je daná ako

$$d(p, p_i) = \|\mathbf{x} - \mathbf{x_i}\| = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2} \tag{3.1}$$

Ak je p_i najbližším bodom od p, alebo je p_i jedným z najbližších bodov od p, tak dostávame vzťah $\|\mathbf{x}-\mathbf{x_i}\| \leq \|\mathbf{x}-\mathbf{x_j}\|$ pre $i \neq j, i, j \in I_n$. Nech $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$ kde $2 \leq n < \infty$ a $\mathbf{x_i} \neq \mathbf{x_j}$ pre $i \neq j, i, j \in I_n$. Región daný vzťahom

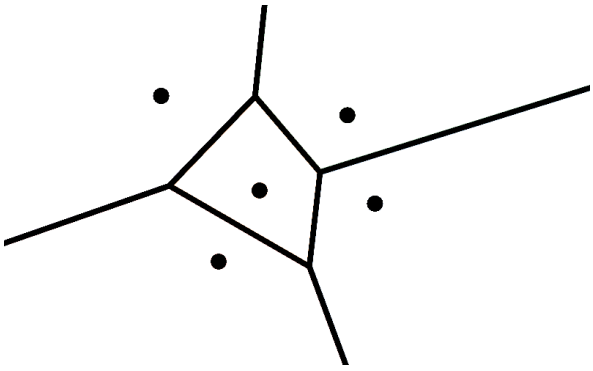
$$V(p_i) = \{\mathbf{x} \mid \|\mathbf{x} - \mathbf{x_i}\| \leq \|\mathbf{x} - \mathbf{x_j}\| prej \neq i, j \in I_n\} \tag{3.2}$$

nazývame *obyčajným rovinným Voronoiovým mnohouholníkom* pridruženým k p_i. Množinu danú vzťahom:

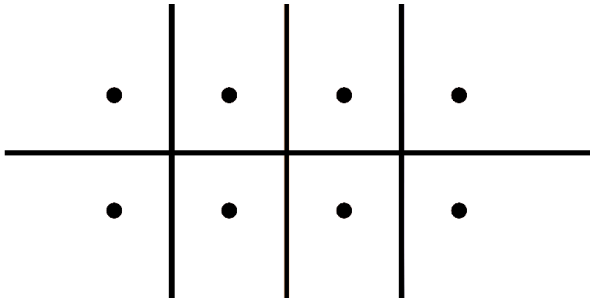
$$\mathcal{V} = \{V(p_1), V(p_2), \ldots V(p_n)\} \tag{3.3}$$

nazývame *obyčajným rovinným Voronoiovým diagramom* generovaným pomocou P. Bod p_i v zápise $V(p_i)$ nazývame ako generujúci bod, alebo generátor i-tého mnohouholníka a množinu $P = \{p_1, p_2, \ldots, p_n\}$ ako generujúcu množinu bodov Voronoiového diagramu $\mathcal{V}$. Ak je $V(p_i) \cap V(p_j) \neq \emptyset$, tak množina $V(p_i) \cap V(p_j)$ udáva tzv. Voronoiovú hranu (ang. Voronoi edge), ktorá môže degenerovať na bod. Túto hranu označujeme $e(p_i, p_j)$, pre $V(p_i) \cap V(p_j)$ a čítame ako *Voronoiová hrana vytvorená pomocou* p_i a p_j. Táto hrana môže byť aj prázdna. Ak hrana $e(p_i, p_j)$ nie je prázdna a nie je ani degenerovaná na bod, tak hovoríme že mnohouholníky $V(p_i)$ a $V(p_j)$ sú susedné. Koncový bod Voronoiovej hrany nazývame ako *Voronoiov vrchol* a značíme q_i. Alternatívne môže byť tento vrchol definovaný ako bod zdieľaný minimálne troma Voronoiovým mnohouholníkmi. Ak potom existuje aspoň jeden vrchol v ktorom sa stretávajú najmenej štyri hrany diagramu $\mathcal{V}$, tak hovoríme že tento diagram je tzv. degenerovaný. Inak diagram $\mathcal{V}$ nazývame nedegenerovaný.

Na obrázku 3.3 je zobrazený degenerovaný diagram a na obrázku 3.2 je zobrazený nedegenerovaný Voronoiov diagram. Degenerovaný diagram sa často objavuje v prípade, kedy sú generujúce body pravidelne rozmiestnené v rovine (podobne ako na obrázku 3.3)



Obr. 3.2: Nedegenerovaný diagram



Obr. 3.3: Degenerovaný diagram

3.2 Vlastnosti

Niektoré zaujímavé vlastnosti Voronoiových diagramov:

- Pre každý vrchol diagramu $q_i \in Q$, existuje unikátny kruh C_i so stredom v bode q_i, taký že pretína ďalšie minimálne tri body z množiny generujúcich bodov P a vo svojom vnútri neobsahuje žiaden z týchto bodov. Tento kruh nazývame *prázdny kruh*.
- Pre každý bod nachádzajúci sa na hrane Voronoiového diagramu, existuje prázdny kruh so stredom v tomto bode, ktorý pretína práve dva generátory Voronoiového diagramu.
- Nech n, n_e, n_v reprezentujú v tomto poradí počet generujúcich bodov diagramu, počet hrán diagramu a počet vrcholov diagramu v $\mathbb{R}^2$. Potom platí: $n_v - n_e + n = 1$
- Priemerný počet hrán mnohouholníka neprekročí 6.

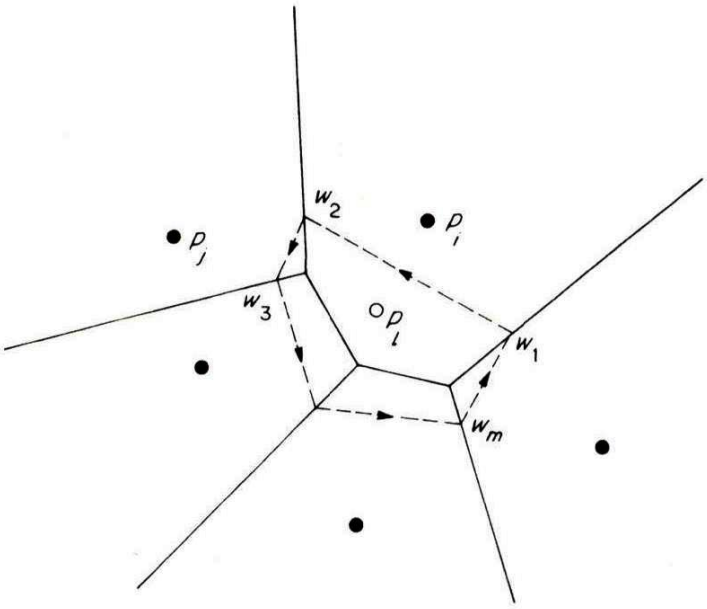
3.3 Algoritmy (metódy) tvorenia

Algoritmov pre tvorbu Voronoiových diagramov existuje veľké množstvo. My sa v tejto kapitole pozrieme na tri základné, medzi ktoré patrí prírastkový (inkrementálny) algoritmus, algoritmus založený na princípe „rozdeľ a panuj“ (ang. *divide and conquer*) a na Fortunov algoritmus.

3.3.1 Prírastkový algoritmus

Je to pomerne jednoduchý algoritmus, ktorý vychádza z počiatočnej fázy v ktorej je zostrojený Voronoiov diagram pre niekoľko málo generátorov (zvyčajne dva alebo tri). Následne sa diagram modifikuje postupným pridávaním ďalších generátorov [28].

Na obrázku 3.4 je pomocou plných čiar zobrazený diagram pre 5 generátorov (čierne body v obrázku). Následne sa pridá do grafu nový generátor (prázdny bod v obrázku) a je potrebné modifikovať diagram vzhľadom k tomuto novo pridanému generátoru. Postupne nachádzame hrany medzi novo pridaným generátorom p_l a už pôvodnými generátormi. Hrany ktoré vzniknú týmto postupom vytvoria mnohouholník (polygón) pre generátor p_l (prerušované čiary v obrázku 3.4). Časti pôvodných hrán, ktoré sú vo vnútri nového mnohouholníka je potrebné odstrániť, čím sa dokončí modifikácia diagramu pre práve pridaný generátor.



Obr. 3.4: Jeden krok prírastkového algoritmu [23]

3.3.2 Algoritmus rozdeľ a panuj

Základným princípom algoritmov rozdeľ a panuj, ktoré sa používajú v rôznych oblastiach, je rozdeliť hlavný problém na podproblémy, ktoré sa riešia samostatne. Výsledok sa získa na konci spojením výsledkov všetkých podproblémov. V prípade použitia tejto metódy pre zostrojenie Voronoiového diagramu je potrebné na začiatku algoritmu zoradiť generátory podľa ich x-ových súradníc, čo je možné vykonať nejakým optimálnym algoritmom pre radenie (napr. pomocou metódy heap sort). Po tomto kroku máme vzostupne zoradený zoznam $P = (p_1, p_2, \ldots, p_n)$, ktorý budeme ďalej používať. V prípade že je počet generátorov nízky (menší ako štyri), tak sa diagram zostrojí priamo. Inak sa množina generátorov rozdelí na dve časti, $P_L = (p_1, \ldots, p_t)$ a $P_R = (p_{t+1}, \ldots, p_n)$, kde t je celé číslo z výsledku po delení $n/2$. Následne sa pre tieto dve množiny, ľavú P_L a pravú P_R, rovnakým spôsobom zostroja Voronoiové diagramy a nakoniec sa čiastočné diagramy spoja do výsledného grafu [28].

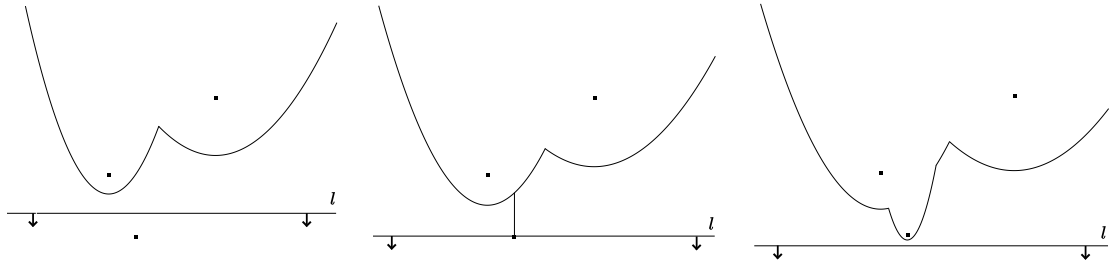
3.3.3 Fortunov algorimtus

Fortunov algoritmus, alebo tiež „zametací algoritmus“ (ang. plane sweep algorithm) je algoritmom pre výpočet Voronoiového diagramu s časovou zložitosťou $O(n) = n * log(n)$. Hlavnou myšlienkou tohto algoritmu je posúvať horizontálnu zametaciu priamku (sweep-line) smerom z hora dole cez rovinu. Počas toho ako sa priamka posúva nadol, dochádza k jej pretínaniu s bodmi v rovine, pričom sa vytvára diagram. Tieto body nazývame bodmi udalostí (ang. *event points*) [23].

Prvým typom týchto bodov sú generátory diagramu, ktoré sa používajú aj v predošlých algoritmoch. Každý generátor F, ktorý sa nachádza nad priamkou vytvára s touto priamkou l parabolu, kde bod F je ohniskom paraboly a priamka l je riadiacou priamkou paraboly. Každý bod na takto definovanej parabole je rovnako vzdialený od ohniska F a od priamky l. Časti takto vzniknutých parabol, ktoré sú najbližšie k zametacej priamke l vytvárajú

ohraničujúcu krivku (beach-line), ktorá je x-monotónna, čiže každá vertikálna priamka ju pretína práve v jednom bode. Táto krivka oddeľuje spodnú časť grafu v ktorej ešte môže dochádzať k zmenám od hornej, v ktorej už žiadne zmeny nenastanú.

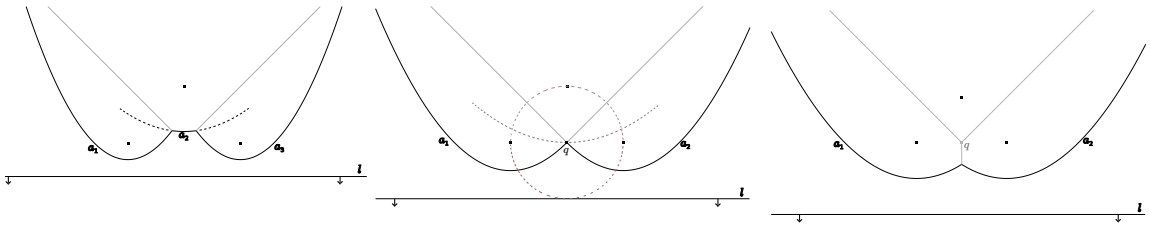
Nové oblúky (časti parabol) na ohraničujúcej krivke vznikajú len v prípade, kedy zametacia krivka pretne generátor. Táto udalosť sa nazýva „site event“, resp. udalosť vzniku paraboly. Priebeh tejto udalosti je zobrazený na obrázku 3.5. V ľavej časti sa zametacia priamka približuje ku generujúcemu bodu, ktorý spôsobí udalosť vzniku paraboly, čo je zachytené v prostrednej časti. Napravo je zobrazený okamih, kedy je už zametacia priamka pod posledným generujúcim bodom a novo-vzniknutá parabola narastá do šírky.



Obr. 3.5: Vznik novej časti ohraničujúcej krivky

Počet oblúkov na ohraničujúcej priamke je maximálne $2n - 1$, keďže každý nový bod vytvorí novú parabolu a rozdelí inú, už existujúcu na dva kúsky [23]. Priesečníky novovzniknutých oblúkov s ohraničujúcou krivkou sú zhodné s bodmi na Voronoiovom diagrame.

Druhým typom udalosti v tomto algoritme je, keď sa existujúci kúsok paraboly zmenší až na bod, a tým zanikne. Túto udalosť nazývame „circle event“, alebo udalosť zániku paraboly. Priebeh tejto udalosti je zobrazený na obrázku 3.6. V ňom, parabola α_2 je tá, ktorá zanikne a paraboly α_1 a α_3 sú jej susedné paraboly, ktoré nemôžu pochádzať z jedného ohniska. V momente kedy parabola α_2 zanikne (scvrkne sa do bodu), majú všetky tri paraboly spoločný priesečník q. Tento priesečník je vtedy rovnako vzdialený od ohniska každej paraboly a taktiež od zametacej priamky l. Dostávame teda kružnicu, ktorej stred je v bode q, prechádza troma ohniskami parabol a jej najnižší bod leží na priamke l. Táto udalosť je jedinou možnosťou ako zanikne kúsok paraboly tvoriaci ohraničujúcu krivku [23]. Bod q, ktorý je stredom kružnice, je taktiež aj vrcholom vo Voronoiovom diagrame.



Obr. 3.6: Zánik časti ohraničujúcej krivky

Celý algoritmus je teda založený na dvoch typoch udalostí, ktoré nastávajú počas toho ako sa priamka pohybuje rovinou a pretína body udalostí. V prípade každej novej udalosti

vzniku paraboly sa kontrolujú novo vzniknuté trojice bodov, či nevytvárajú kružnicu, ktorá by mala za následok vznik ďalšej udalosti. Udalosti ktoré takto vznikajú sa zaraďujú do rady na spracovanie podľa ich y-ovej súradnice. Tento postup sa opakuje pokiaľ priamka nedosiahne koniec roviny, kedy je ešte potrebné zakončiť hrany ktoré doposiaľ dokončené neboli.

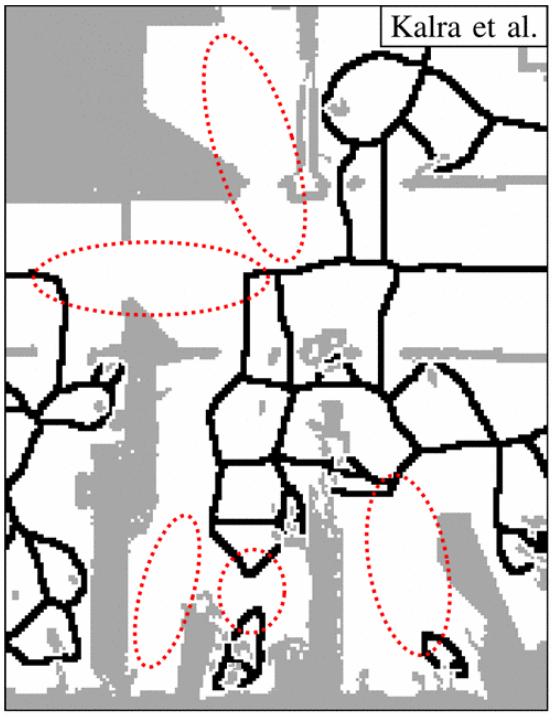
3.4 Metódy plánovania cesty pomocou Voronoiových diagramov

Analytické metódy pre výpočet Voronoiových diagramov sú síce efektívne a niektoré optimálne, no pre použitie v robotike nie sú veľmi vhodné, nakoľko väčšina robotických operačných systémov využíva pre plánovanie ciest diskrétne mapy vo forme obrázkov. V týchto bit-mapových obrázkoch predstavujú tmavé pixely prekážky a svetlé zasa voľný priestor, kadiaľ môže robot prechádzať.

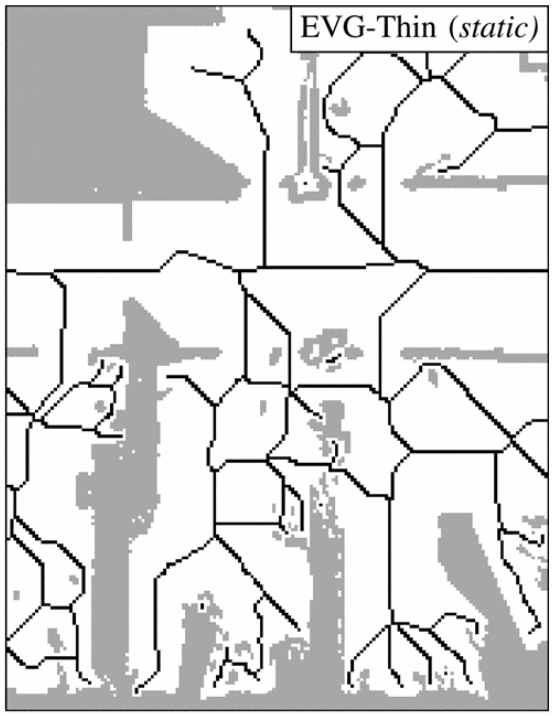
Rozdiel je aj v samotnom Voronoiovom diagrame, ktorý je v diskrétnom priestore odlišný od toho v spojitom. Nakoľko v spojitom priestore je hrana zanedbateľne úzka, keďže je to len množina priamok a kriviek, zatiaľ čo v spojitom priestore musí mať hrana šírku aspoň jednej bunky mapy. Voronoiov diagram v spojitom priestore potom označujeme ako zovšeobecnený Voronoiov diagram (Generalised Voronoi diagram - GVD). Jednou z možností generovania GVD, je použitie grafického hardvéru, ktorý to dokáže vypočítať vo veľmi krátkom čase. Avšak, použitie v robotike nie je veľmi reálne, pretože hardvér robota je často veľmi obmedzený a grafické procesory tam nemajú svoje miesto. Okrem toho existujú aj hardvérovo nezávislé algoritmy, ktoré sú dostatočne rýchle pre výpočet GVD z diskrétnej mapy. Vo všeobecnosti tieto algoritmy prechádzajú mapu a pre každú jej bunku vypočítavajú vzdialenosť k najbližšej prekážke.

Prvým z týchto algoritmov je Brushfire algoritmus. Je to obdoba Dijkstrovho algoritmu pre plánovanie cesty, ktorá spracováva bunky v prioritnej fronte. Jeho popis spolu s pseudokódom je v kapitole 4.1.1. Ďalší zo spôsobov predstavuje Kalra a spol. [26], pričom zavádzajú identifikátory jednotlivých prekážok, ktoré sú unikátne prideľované každému zoskupeniu buniek v mape. Potom ak podľa tejto identifikácie majú dve susedné bunky rozdielne najbližšie prekážky , sú obe bunky označené ako súčasť GVD. Nevýhodou tohto prístupu je to, že sa vytvára diagram, ktorého hrany sú dve bunky široké. Taktiež tento spôsob nefunguje vo vnútri konkávnych objektov, akými sú napríklad miestnosti, alebo chodby, čo je vyobrazené na obrázku 3.7 v červených zakrúžkovaných častiach. Táto vlastnosť deštruuje súvislosť Voronoiového diagramu a tým znemožňuje plánovanie cesty cez tieto úseky mapy, čo je problematické najmä pri plánovaní ciest vo vnútri budov.

Zhang a Suen [30] prišli s myšlienkou pre vytváranie GVD nad statickou mapou kde postupne zaniká voľný priestor v mape, pričom začínajú v miestach, ktoré sú najbližšie pri prekážkach. Každá bunka je skúmaná množinou tzv. stenčovacích pravidiel, ktoré rozhodujú o tom, či daná bunka môže zaniknúť alebo nie. Ich myšlienku použil Beeson, ktorého implementácia známa ako „EVG-Thin“ vytvára grafy, ktoré sú v porovnaní s tými od Kalra tenšie, no občas sa objavia cesty širšie ako jedna bunka a v niektorých prípadoch sú hrany GVD len aproximáciou tých skutočných hrán [27]. Avšak táto metóda sa už dokáže vysporiadať s miestnosťami, kde mal algoritmus od Kalra problémy. Výsledok je zobrazený na obrázku 3.8.



Obr. 3.7: Diagram vytvorený metódou od Kalra a spol. [27]



Obr. 3.8: Diagram vytvorený metódou EVG-Thin [27]

Z predchádzajúcich dvoch metód vychádza Lau a spol. [27], ktorí vyvinuli spoľahlivú metódu ktorá generuje tenké hrany Voronoiového diagramu, ktoré už nie sú len aproximáciou reálnych GVD hrán a k ich vytvoreniu nepotrebuje identifikovať prekážky v mape.

Kapitola 4

Návrh zásuvného modulu

Pre účely navigácie a teda v našom prípade aj pre výpočet Voronoiového diagramu sa v robotike využívajú diskrétne obrazové mapy. V robotickom operačnom systéme na to slúži mapa, tzv. *costmap*, ktorá pre každú bunku mapy obsahuje ocenenie, ktoré určuje či je daná bunka voľná pre prechod robota alebo obsadená nejakou prekážkou. Ocenenia v tejto mape nadobúdajú hodnoty z intervalu od 0 do 1 vrátane oboch hraničných hodnôt a špeciálnu hodnotu -1. Bunka mapy, ktorej ocenenie je nulové predstavuje voľnú, neobsadenú bunku. Miesta ocenené hodnotou -1 znamenajú, že oblasť nie je v týchto miestach preskúmaná, teda hodnota ocenenia je neznáma. Všetky ostatné bunky su považované za obsadené. Hodnoty z tejto mapy sa pre účely plánovania prevedú na binárnu mapu, ktorá bude obsahovať len hodnoty 1 pre obsadené bunky a 0 pre bunky voľné. Tá sa potom vloží na vstup algoritmu pre plánovanie cesty.

Plánovanie cesty pomocou Voronoiových diagramov pozostáva z niekoľkých hlavných častí. Prvou z nich je vypočítanie Voronoiového diagramu pre danú mapu, v ktorej sa bude robot pohybovať. V ďalšej, nie menej podstatnej časti, je potrebné nájsť cestu z miesta, kde sa robot aktuálne nachádza na miesto ktoré je súčasťou Voronoiového diagramu (bod príchodu na diagram). Potom je potrebné nájsť cestu naprieč diagramom a nakoniec spojnicu medzi posledným bodom na ceste diagramom (bod odchodu z diagramu) a cieľovým bodom. Metódy pre všetky tieto štyri fázy si popíšeme v tejto kapitole.

4.1 Výpočet Voronoiového diagramu

Existuje niekoľko algoritmov pre výpočet Voronoiového diagramu nad diskrétnou mapou, pričom niektoré z nich boli predstavené v kapitole 3.4. Pre tento zásuvný modul boli zvolené dve metódy, ktorých výsledky sú čiastočne odlišné a každá z nich je vhodnejšia pre iný typ mapy. V nasledujúcich častiach si popíšeme ich algoritmy, výhody a nevýhody, a taktiež si ukážeme výsledky po spracovaní vstupnej mapy.

4.1.1 Brushfire algoritmus

Prvým zo spomínaných algoritmov je tzv. *Brushfire algoritmus*, ktorý popisuje aj Bräunl [24]. Jeho prístup by bolo možné rozdeliť na dve časti, kde prvou z nich je klasifikácia mapy a druhou sú prechody tejto mapy. Kompletný pseudokód je v algoritme 1.

Počiatočným krokom tejto metódy je klasifikácia prekážok v mape, tak aby každá z nich mala priradený vlastný identifikátor, resp. farbu a taktiež je potrebné unikátnu farbu priradiť aj hraniciam mapy. Zároveň sa každej bunke prekážky nastaví hodnota vzdialenosti na 1,

čo znamená že bunka je prekážkou a bude sa to využívať v ďalších fázach algoritmu. Časť klasifikácie možno vykonať pomocou záplavového (*flood fill*) algoritmu, ktorý je popísaný nižšie.

```
Input: binary map with obstacles
Result: map with Voronoi diagram
classifyObstacles();
classifyBorders;
iteration ← 2;
while something changed do
    foreach cell : map do
        if cell is free and not processed then
            if cell is adjacent to already processed cell then
                cell.color ← adj_cell.color;
                cell.dist ← iteration;
            end
            if cell.colorChanged() then
                cell.voronoi ← true;
            end
        end
    end
    iteration++;
end
foreach cell : map do
    if cell.dist == top_cell.dist then
        if cell.color != top_cell.color then
            cell.voronoi ← true;
        end
    end
    if cell.dist == right_cell.dist then
        if cell.color != right_cell.color then
            cell.voronoi ← true;
        end
    end
end
```

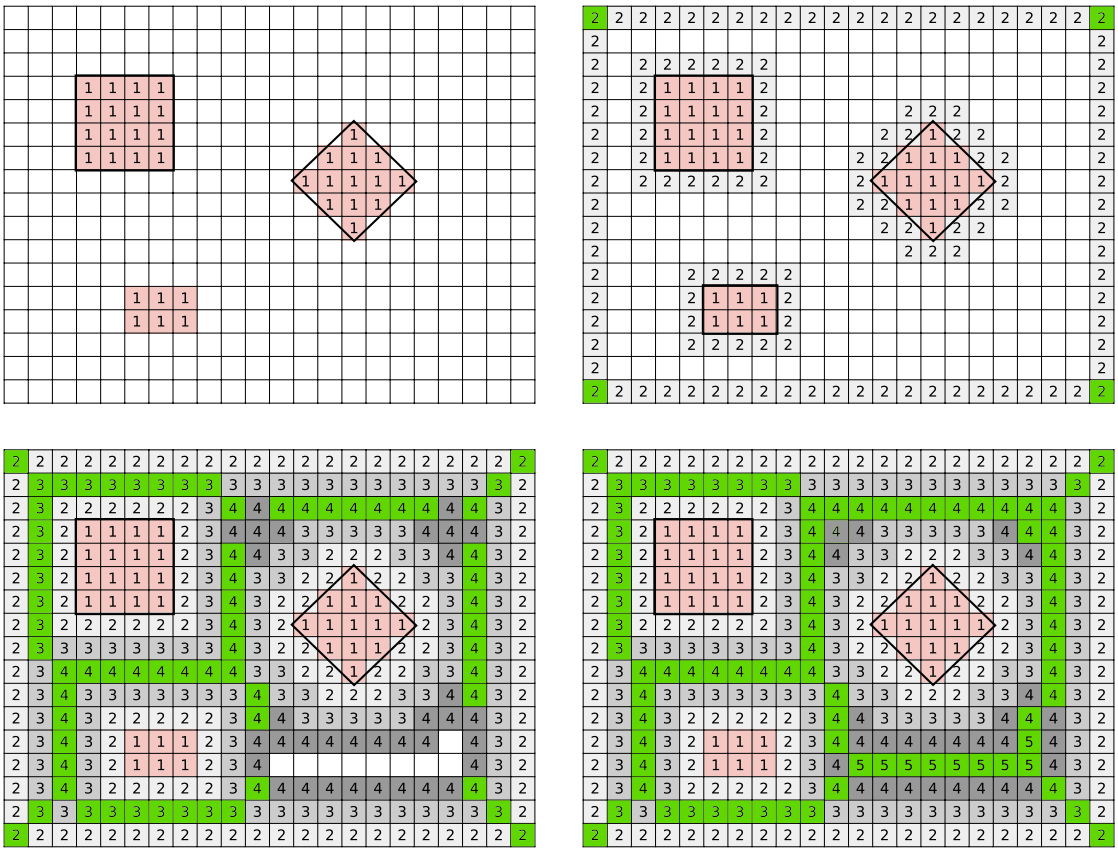
Algoritmus 1: Brushfire algoritmus pre výpočet Voronoiovho diagramu

Ďalšou fázou v brushfire algoritme je opakované prechádzanie mapy, dokým nebudú spracované všetky bunky mapy. V prvom cykle sa spracovávajú len tie bunky, ktoré sú susedné k bunkám prekážok. V každom ďalšom prechode sa spracovávajú bunky ktoré sú susedné k tým už spracovaným. Aktuálne modifikovanej bunke sa nastaví hodnota vzdialenosti o jednu väčšia než má susedná bunka a farba zhodná s tou susednou. Pokiaľ nastane situácia, že farba bunky bude minimálne dvakrát zmenená podľa farieb susedných buniek, potom ju môžeme označiť ako bod na Voronoiovom diagrame.

Poslednou fázou algoritmu je ešte jeden prechod všetkých buniek mapy. Pri tomto prechode sa porovnáva aktuálne skúmaná bunka so susednou, ktorá je nad ňou a napravo vedľa nej. Ak má aspoň jedna z dvojíc (aktuálna–horný sused alebo aktuálna–pravý sused)

zhodné hodnoty vzdialenosti a rozdielne farby, tak je aktuálna bunka označená ako bod na Voronoiovom diagrame.

Priebeh algoritmu pre jednoduchú mapu je možné vidieť na obrázku 4.1, kde sú načerveno vyfarbené a hrubou čiarou ohraničené prekážky a v každej bunke je zapísaná jej hodnota vzdialenosti. Bunky vyfarbené na zeleno predstavujú body Voronoiového diagramu.



Obr. 4.1: Priebeh brushfire algoritmu

Záplavový (*flood fill*) algoritmus, je možné po malom prispôsobení využiť v tomto prípade na klasifikáciu prekážok. Existuje jeho niekoľko rôznych variant, ktoré využívajú zásobník, frontu či rekurzívne volania. Pre náš program sme zvolili verziu, ktorá pre ešte nespracované bunky využíva frontu a rozširovanie farby (typu) prekážky sa deje do oboch smerov. Algoritmus postupne prejde všetky bunky mapy a každú z nich na začiatku vloží do fronty, z ktorej ju následne vyberá, dokiaľ fronta obsahuje nejaké prvky. Ak je bunka ešte nespracovaná, teda ešte nemá priradenú žiadnu farbu, priradí sa jej taká, ktorá ešte nebola použitá. Následne sa skúmajú bunky ktoré sú v rovnakom riadku ako aktuálna bunka. Najprv sa spracovávajú tie naľavo od aktuálnej. Ak je bunka prekážkou, tak sa jej priradí rovnaká farba a propagácia doľava pokračuje, až pokiaľ nenarazí na voľnú bunku, alebo na okraj mapy. Index poslednej zafarbenej bunky sa uloží pre ďalšie spracovanie. Obdobne sa farba buniek prekážok rozširuje aj do pravej strany. Po skončení propagácie do oboch strán sa ešte preskúmajú horná a dolná susedná bunka každej bunky od minimálneho ľavého indexu, po maximálny pravý index. Overujú sa rovnaké podmienky ako v prvom kroku a ak sú splnené, tak je bunka vložená do fronty ešte nespracovaných buniek.

Algoritmus brushfire ktorý popisuje Bräunl je pomerne jednoduchý no dosť efektívny a presný. No žiaľ nedokáže sa vysporiadať so všetkými rozloženiami mapy. V tých mapách kde je vykreslená len jedna miestnosť s prekážkami v nej, funguje brushfire bez problémov, ale ak mu predložíme mapu celej budovy, kde sú prechody medzi miestnosťami a úzke miestnosť, či chodby, tak tam nedokáže vytvoriť Voronoiov diagram. Preto bude v zásuvnom module implementovaný aj algoritmus od Lau a spol., ktorý s vytváraním diagramu v chodbách a úzkych prechodoch nemá problém ale zasa v miestnostiach plných prekážok, ktoré nie sú štvoruholníkové vytvára hrany, ktoré by tam byť nemali. Jeho prístup si popíšeme v ďalšej časti.

4.1.2 Algoritmus od Lau

Algoritmus pre výpočet Voronoiového diagramu, ktorý predstavuje Lau a spol. [27] je možné rozdeliť do troch základných častí. V prvej z nich sa vypočítavajú vzdialenosti každej bunky v mape k jej najbližšej prekážke. Následne sa kontrolou vlastnosti buniek v mape určí, ktoré z nich môžu byť súčasťou Voronoiového diagramu a v poslednej fáze sa vykoná odstránenie tých bodov na diagrame, ktoré vytvárajú zbytočne široké hrany.

Výsledkom prvej časti tohto algoritmu, je tzv. mapa vzdialeností (*distance map*), kde má každá bunka určenú vzdialenosť k jej najbližšej prekážke a v tomto prípade aj jej pozíciu. V inicializačnej fáze tejto časti, sa na základe dát zo vstupnej mapy priradí každej bunke vzdialenosť a typ prekážky podľa toho či je, alebo nie je prekážkou. Voľným bunkám je nastavená hodnota vzdialenosti na maximum, keďže ešte nie je známa a typ prekážky je nedefinovaný. Naopak u buniek obsadených prekážkou je vzdialenosť rovná nule a bunky odkazujú samé na seba ako na prekážku. Tieto obsadené bunky sa tiež spolu so svojou vzdialenosťou vložia do prioritnej fronty, odkiaľ budú v ďalšej fáze vyberané. Pseudokód tejto časti algoritmu je v algoritme 2.

```
foreach cell : map do
    if cell is obstacle then
        cell.dist ← 0;
        cell.obst ← cell;
        queue.insert(0, cell);
    else
        cell.dist ← ∞;
        cell.obst ← ∅;
    end
end
```

Algoritmus 2: Inicializačná časť algoritmu pre výpočet vzdialeností

V druhej fáze prvej časti sa z fronty postupne vyberajú prvky s najnižšou hodnotou vzdialenosti a spracovávajú sa ich susedné bunky. Pre každú priľahlú bunku sa vypočíta jej vzdialenosť k prekážke bunky ktorá bola vybraná z fronty. Ak je vypočítaná vzdialenosť menšia ako tá ktorá je uložená v susediacej bunke, tak sa spolu s odkazom na prekážku aktualizujú hodnoty susednej bunky a tá sa vloží do prioritnej fronty. Vkladanie sa uskutoční len vtedy, ak sa bunka ešte nenachádza vo fronte. Inak sa len aktualizujú hodnoty prvku už umiestneného vo fronte. Pseudokód tejto časti je v algoritme 3.

Druhou časťou celého algoritmu je výpočet Voronoiového diagramu. Ten prebieha tak, že sa postupne prejdú všetky bunky mapy a ich susedné bunky a ak sú splnené určité

```
while queue not empty do
    cell ← queue.pop();
    foreach neighbor : cell do
        d ← dist(cell.obst, neighbor);
        if d < neighbor.dist then
            neighbor.dist ← d;
            neighbor.obst ← cell.obst;
            queue.insert(d, neighbor);
        end
    end
end
```

Algoritmus 3: Algoritmus výpočtu vzdialeností od prekážok

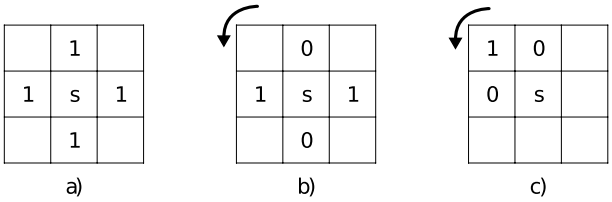
podmienky tak je bunka označená za súčasť diagramu. Pseudokód tejto časti je v algoritme 4. Prvou podmienkou, ktorá musí byť splnená je to, že aspoň jedna bunka z aktuálne skúmanej dvojice (hlavná bunka *cell* a jej susedná bunka *neigh*) musí byť od prekážky vzdialená o viac ako jednu bunku. Ďalej je potrebné aby bol definovaný odkaz na prekážku bunky *neigh* a tá aby nebola zhodná s prekážkou bunky *cell*. Zároveň je potrebné aby prekážka bunky *cell* nepatrila do okolia bunky *neigh*. Ak by tieto podmienky neboli splnené, tak by mohli vznikať v diagrame hrany, ktoré tam v skutočnosti nepatria.

```
foreach cell : map do
    foreach neigh : cell do
        if (cell.dist > 1 ∨ neigh.dist > 1)
          ∧ neigh.obst ≠ ∅ ∧ neigh.obst ≠ cell.obst
          ∧ cell.obst ∉ neighborhood of neigh.obst then
            c_no ← dist(cell, neigh.obst);
            n_co ← dist(neigh, cell.obst);
            if c_no ≤ n_co then
                cell.voronoi ← true;
            end
            if n_co ≤ c_no then
                neigh.voronoi ← true;
            end
        end
    end
end
```

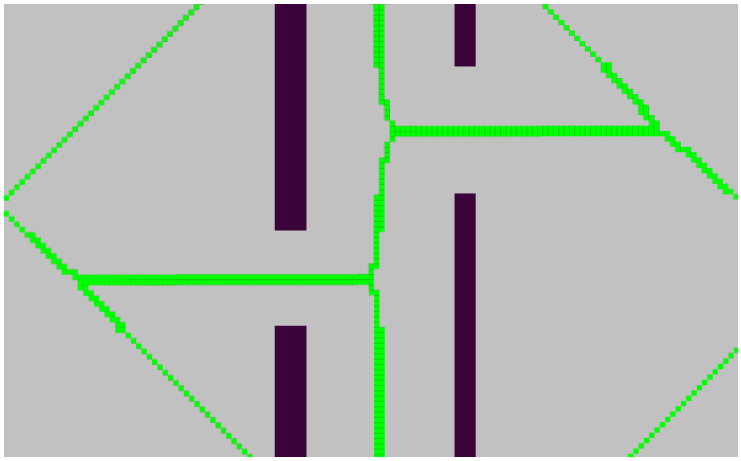
Algoritmus 4: Algoritmus výpočtu Voronoiového diagramu

Poslednou, treťou, časťou algoritmu je fáza čistenia diagramu a to tak aby boli odstránené bunky ktoré vytvárajú zbytočne široké hrany. Ešte pred tým sa však prejde celá mapa a do Voronoiového diagramu sa doplnia bunky ktoré v ňom nie sú, ale sú nimi obklopené ako na obrázku 4.2 v časti **a)**, kde bunka s hodnotou 1 predstavuje bod diagramu, bunka s 0 bunku ktorá nie je súčasťou diagramu a prostredná je práve skúmaná. Týmto krokom sa zamedzí chybovým miestam, ktoré vznikajú kvôli obmedzenému, konečnému, rozlíšeniu vstupnej mapy.

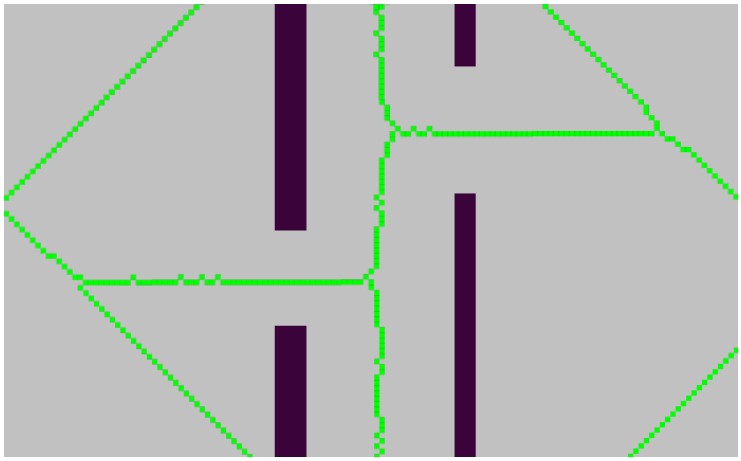
Samotné čistenie prebieha tak, že všetky bunky tvoriace hrany Voronoiového diagramu sa vložia do fronty, v ktorej sú zoradené podľa vzdialenosti k najbližšej prekážke. Potom sa postupne vyberajú, počnúc tými s najnižšou vzdialenosťou. Ak vlastnosti bunky a jej okolia nezodpovedajú ani jednému z troch vzorov na obrázku 4.2, pričom vzory **b)** a **c)** sa môžu otáčať o 90°, tak bunka bude odstránená z diagramu. Ukážka diagramu pred čistením a po ňom je zobrazená na obrázkoch 4.3 a 4.4.



Obr. 4.2: Vzory pre odstraňovanie prebytočných buniek z diagramu



Obr. 4.3: Časť Voronoiovho diagramu pred fázou čistenia



Obr. 4.4: Časť diagramu po dokončení fázy čistenia

4.2 Hľadanie cesty mimo diagramu

Ďalšou fázou v procese plánovania cesty v tomto zásuvnom module je, po vypočítaní Voronoiového diagramu, nájdenie prepojenia, teda časti cesty ktorá prepojí počiatočnú pozíciu s bodom na diagrame (bod príchodu). Obdobne sa rovnaký algoritmus použije pre nájdenie prepojenia medzi cieľovou pozíciou a Voronoiovým diagramom (bodom odchodu). Pre tieto účely bol zvolený *Dijkstrov algoritmus*.

Na jeho začiatku sa každej bunke mapy, nastaví hodnota vzdialenosti od počiatočnej pozície na maximum a referencia na rodičovskú bunku sa nastaví ako neznáma. Výnimkou je počiatočná bunka, ktorej hodnota bude 0 a táto bunka sa vloží do fronty, v ktorej budú bunky zoradené podľa vzdialenosti. Nasleduje cyklus, v ktorom sa z fronty vyberie bunka s najnižším ohodnotením a spracuje sa. Celý cyklus sa končí vtedy, keď je bunka bodom na Voronoiovom diagrame, prípadne ak je to cieľová bunka (táto situácia nastane ak sa cieľ nachádza k počiatku bližšie ako hrana diagramu). Spracovanie bunky vybratej z fronty spočíva v tom že sa nastavia hodnoty vzdialeností susedným bunkám, prípadne ak už sú nastavené, tak sa môžu aktualizovať. Hodnota vzdialenosti susednej bunky je rovná súčtu hodnoty vzdialenosti hlavnej bunky s hodnotou vzájomnej vzdialenosti buniek. Ak je vypočítaná hodnota menšia ako tá, ktorá je uložená v bunke, tak sa vykoná aktualizácia hodnoty. Zároveň sa aktualizuje referencia na rodičovskú bunku, ktorá sa bude využívať pri vytváraní cesty. Po skončení cyklu sa podľa referencii na rodičovské bunky nájde cesta z počiatočného bodu do cieľového.

4.3 Hľadanie cesty naprieč diagramom

Poslednou fázou v plánovaní je nájdenie cesty naprieč Voronoiovým diagramom z bodu príchodu naňho, do bodu odchodu z neho. Pre túto úlohu bol zvolený algoritmus A*, ktorý pri hľadaní najkratšej cesty berie do úvahy cieľový bod. Jeho podstata je rovnaká ako u Dijkstrovho algoritmu, ale líši sa v počítaní vzdialenosti buniek, kde sa ešte pripočítava hodnota heuristiky v danom bode. Heuristika je určená ako Euklidovská vzdialenosť medzi daným bodom a cieľovým bodom. V tomto algoritme sa využívajú dva zoznamy: zoznam *open*, ktorý obsahuje ešte nespracované body a zoznam *close*, ktorý s už spracovanými uzlami. Taktiež sa používajú dve pomocné polia s hodnotami vzdialeností buniek mapy a pole s odkazmi na rodičovské bunky.

Kapitola 5

Popis implementácie modulu

V tejto kapitole si popíšeme triedy implementovaného balíčka a najpodstatnejšie metódy z neho, taktiež si povieme aké sú možnosti nastavenia plánovača a v poslednej časti sa pozrieme na zobrazovanie dát z plánovača vo vizualizéri pre robotický operačný systém.

5.1 Implementácia

Implementácia zásuvného modulu je v robotickom operačnom systéme možná v dvoch programovacích jazykoch, ktorými sú Python a C++. Prvý zo spomínaných je interpretovaným jazykom, čiže pred spustením kódu sa nevykonáva jeho preklad do strojového kódu ale príkazy sa vykonávajú postupne tak, ako sú napísané v zdrojovom texte. Výhodou tohto typu jazykov je, že nie je potrebné po každej, hoci i malej, zmene v programe vykonávať preklad, ktorý môže byť časovo náročný. Na druhej strane stojí rýchlosť výpočtu, keďže Python je dynamicky typovaný jazyk, teda kontroly dátových typov sa vykonávajú počas behu programu, čo zaberá nemalé množstvo výpočtového času. Z tohto dôvodu je skôr vhodný pre prototypovanie, prípadne pre použitie v systémoch s dostatočným výpočtovým výkonom. Vývoj v jazyku C++ je o niečo zdĺhavejší, keďže po každej zmene v zdrojovom kóde je potrebné program nanovo preložiť, avšak následné vykonávanie kódu je efektívnejšie v porovnaní s jazykom Python. Aj preto je náš zásuvný modul pre plánovanie cesty implementovaný v jazyku C++.

Celý balíček nášho globálneho plánovača pozostáva z dvoch tried, ktoré sú obsiahnuté v mennom priestore `voronoi_path_planner`. Hlavná trieda balíčka, ktorá obsahuje dôležité funkcie pre plánovanie cesty je `voronoi_path_planner::VoronoiPathPlanner` a využíva objekty triedy `voronoi_path_planner::Pixel`. Nevyhnutnou súčasťou pri implementácii triedy globálneho plánovača je použitie makra `PLUGINLIB_EXPORT_CLASS()`, ktoré vytvorenú triedu zaregistruje a exportuje tak, aby ju robotický operačný systém rozpoznal ako variantu pre globálny plánovač cesty a aby ju v prípade použitia vedel dynamicky načítať. Ďalšou podmienkou pre hlavnú triedu globálneho plánovača je, že musí implementovať rozhranie `nav_core::BaseGlobalPlanner` z balíčka `nav_core`, ktorý je jednou z hlavných častí ROS a využíva sa pre účely navigácie. Spomínané rozhranie poskytuje metódu `initialize()` a metódu `makePlan()`, ktoré sú, mimo iných, implementované v našej triede.

Metóda `initialize()` sa volá pri spúšťaní robota, kedy sa inicializuje aj globálny plánovač. Počas inicializácie sa získavajú atribúty mapy, ako napríklad jej výška, šírka a rozlíšenie a taktiež sa z nej získavajú dáta o prekážkach a voľných miestach. Ďalej sa vo fáze inicializácie nastavujú atribúty tém (*topics*), ktoré sa budú používať neskôr a ak boli zadané aj

parametre pre globálny plánovač, tak sa príslušne nastavia. Na základe tohto nastavenia sa následne zavolá ďalšia metóda, ktorá pre zadanú mapu vypočíta Voronoiov diagram.

Metóda `makePlan()` sa je volaná v prípade, kedy je potrebné vypočítať novú cestu z aktuálnej do cieľovej pozície robota. Pri tomto hľadaní najvhodnejšej cesty zo štartu k cieľu pomocou Voronoiovho diagramu sa využíva A* a Dijkstrov algortimus, nezávisle od toho, akým spôsobom bol diagram vypočítaný.

Objekty druhej triedy, `voronoi_path_planner::Pixel`, sa využívajú pre uloženie informácii o bunkách mapy. Každá bunka mapy obsahuje informáciu o tom či je alebo nie je prekážkou, a či je alebo nie je bodom na Voronoiovom diagrame. V prípade, že bunka nie je prekážkou, tak obsahuje informáciu o tom ako ďaleko je vzdialená najbližšia prekážka, jej typ, a taktiež pozícia v mape. Všetky tieto atribúty a metódy pre prácu s nimi sú potrebné pre výpočet Voronoiovho diagramu.

5.2 Použitie a prispôsobenie

Pre využitie nášho globálneho plánovača je potrebné správne nastaviť parameter globálneho plánovača v balíčku `move_base`, čo sa zvyčajne nastavuje v súbore `move_base.launch.xml`. V ňom treba upraviť alebo pridať parameter s názvom `base_global_planner` na hodntou `"voronoi_path_planner/VoronoiPathPlanner"`. Text ktorý je nutné pridať v prípade, že sa doteraz využíval implicitný plánovač je následovný:

```
<param name="base_global_planner"
       value="voronoi_path_planner/VoronoiPathPlanner"/>
```

Parameter, ktorý rozhoduje o variante výpočtu Voronoiového diagramu, teda či sa použije algoritmus od Bräunl alebo od Lau je definovaný v implementovanom balíčku v priečinku `param/`, konkrétne v súbore `voronoi_path_planning_params.yaml`. Názov tohto parametra je `voronoi_planner_method` a hodnoty ktoré môže nadobúdať sú `"Braunl"`, alebo `"Lau"`. Druhá z nich je implicitná možnosť a použije sa aj v prípade, ak bol parameter nastavený nesprávne. Práve tento parameter určuje, ktorá metóda sa zavolá v metóde `initialize()`, ktorú sme spomínali vyššie.

Druhým parametrom ktorý ovplyvňuje výpočet Voronoiového diagramu hľadanie cesty je `inflation_radius`. Tento parameter zabezpečí akoby zväčšenie prekážok o hodnotu zadanú v tomto parametri. Dôvod, ktorý vedie k tomuto „nafukovaniu“ prekážok je ten, že globálne plánovače hľadajúce najkratšiu možnú cestu k cieľu, produkujú cesty ktoré vedú veľmi blízko popri hranách prekážok a v dôsledku čoho dochádza často ku kolíziám robota s týmito prekážkami. Predvolená hodnota tohto parametra je pre robota Turtlebot nastavená na 0,5 metra. Pri použití nášho plánovača, je možné tento parameter vynulovať, keďže cesty vedú vo väčšine prípadov stredom medzi dvoma prekážkami a teda riziko kolízie je minimálne. Nastavenie hodnoty tohto parametra je možné pridaním následujúceho riadku do súboru `move_base.launch.xml`

```
<param name="global_costmap/inflation_layer/inflation_radius" value="0.0"/>
```

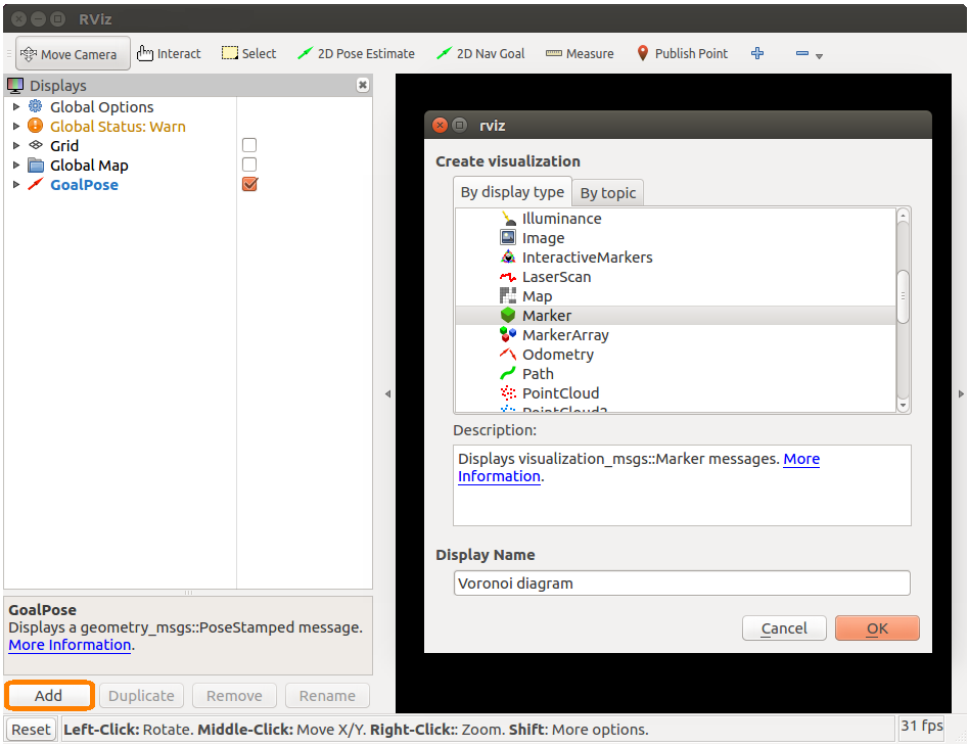
5.3 Zobrazovanie dát v RViz

Implementovaný globalný plánovač obsahuje okrem iného aj dve témy *(topic)*, na ktoré sa vysielajú dáta v ňom vyprodukované. Prvou témou je `Voronoi_diagram`, kam sa vysie-

lajú správy typu `visualization_msgs::Marker`. Dáta publikované v týchto správach sú súradnice bodov, ktoré dohromady vytvárajú Voronoiov diagram.

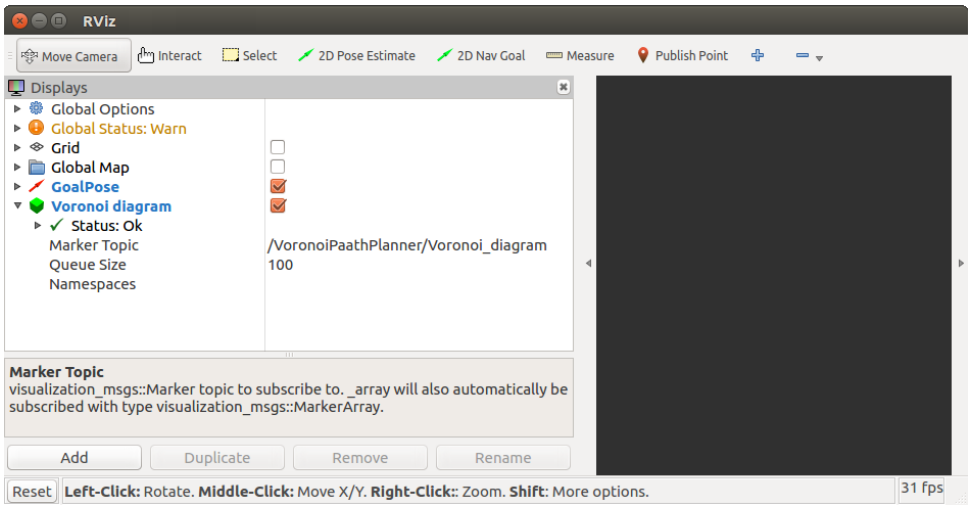
Druha téma s názvom `Voronoi_full_path` slúži na zobrazenie celej výpočítanej cesty z miesta štartu robota, až do zvoleného cieľa. Správy tejto témy sú typu `nav_msgs::Path` a celá cesta pozostáva z bodov, pozícii, ktorými by mal robot prejsť na ceste k cieľu.

Pre zobrazenie týchto vyprodukovaných dát vo vizualizéri RViz pre robotický operačný systém, je potrebné pridať novú položku, do zoznamu zobrazovaných dát. To vykonáme kliknutím na tlačidlo *Add* v ľavom dolnom rohu, ako je vidieť na obrázku 5.1. Následne si zo zoznamu v novom okne vyberieme, aký druh dát chceme zobrazovať. Pre vykreslenie Voronoiového diagramu zvolíme typ *Marker*, a pre vypočítanú cestu zasa typ *Path*. Potom si novo pridanú položku môžeme pomenovať v spodnej časti okna, aby sme sa v zobrazovaných dátach dobre orientovali. Posledným krokom pre správne zobrazovanie požadovaných dát



Obr. 5.1: Prvá časť pridávania novej položky v RViz

je nastavenie témy, ktorej dáta budú vykresľované. Tému vyberáme zo zoznamu v riadku s atribútom *Marker Topic*, prípadne *Topic*. Pre zobrazenie bodov Voronoiového diagramu vyberáme možnosť `/VoronoiPathPlanner/Voronoi_diagram`, ako je zobrazené na obrázku 5.2. Obdobne zvolíme tému pre zobrazenie vypočítanej cesty, pričom téma ktorú chceme zobraziť je `/VoronoiPathPlanner/Voronoi_full_path`.



Obr. 5.2: Druhá časť pridávania novej položky v RViz

Kapitola 6

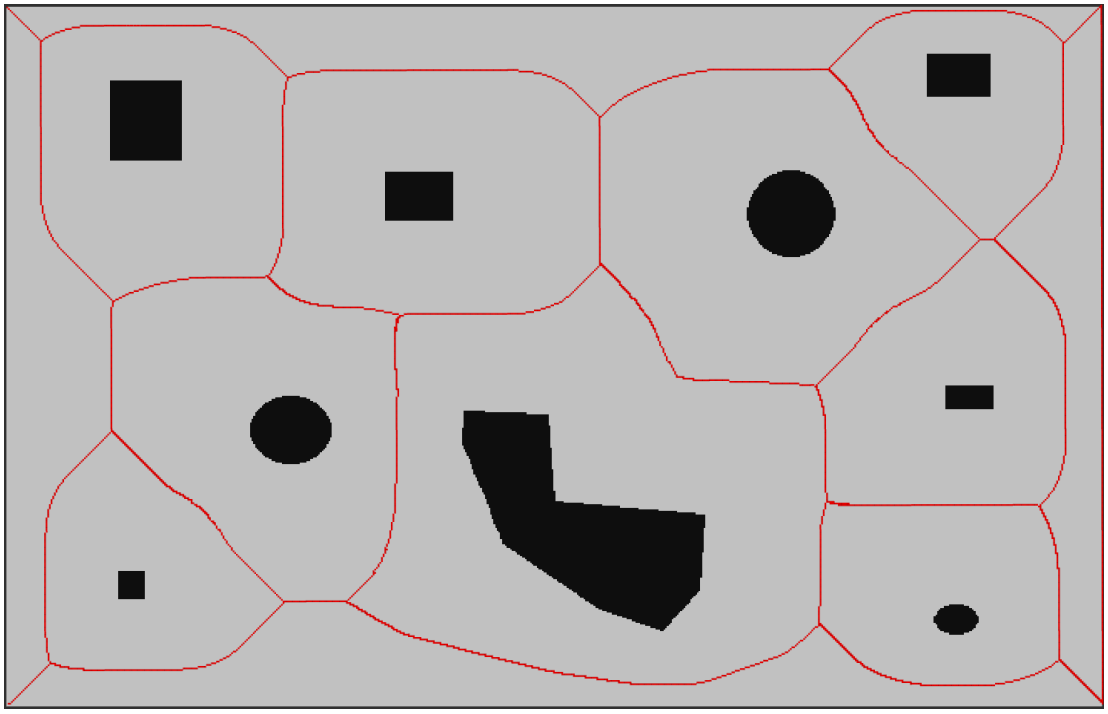
Testovanie a zhodnotenie

Táto kapitola obsahuje v prvej časti porovnanie Voronoiových diagramov vytvorených nad rovnakou mapou pomocou prístupu od Bräunl a Lau. V druhej časti zasa porovnanie výsledkov plánovania cesty v piatich rôznych mapách za použitia troch plánovačov, z toho je jeden náš a dva implementované v ROS. Hodnotiť sa bude čas výpočtu cesty, jej dĺžka a taktiež ako je cesta vzdialená od prekážok. Naplánované cesty a vypočítané Voronoiové diagramy boli vykresľované pomocou vizualizéru RViz. Na konci kapitoly je tabuľka 6.1, v ktorej sú zhrnuté výsledky porovnávania.

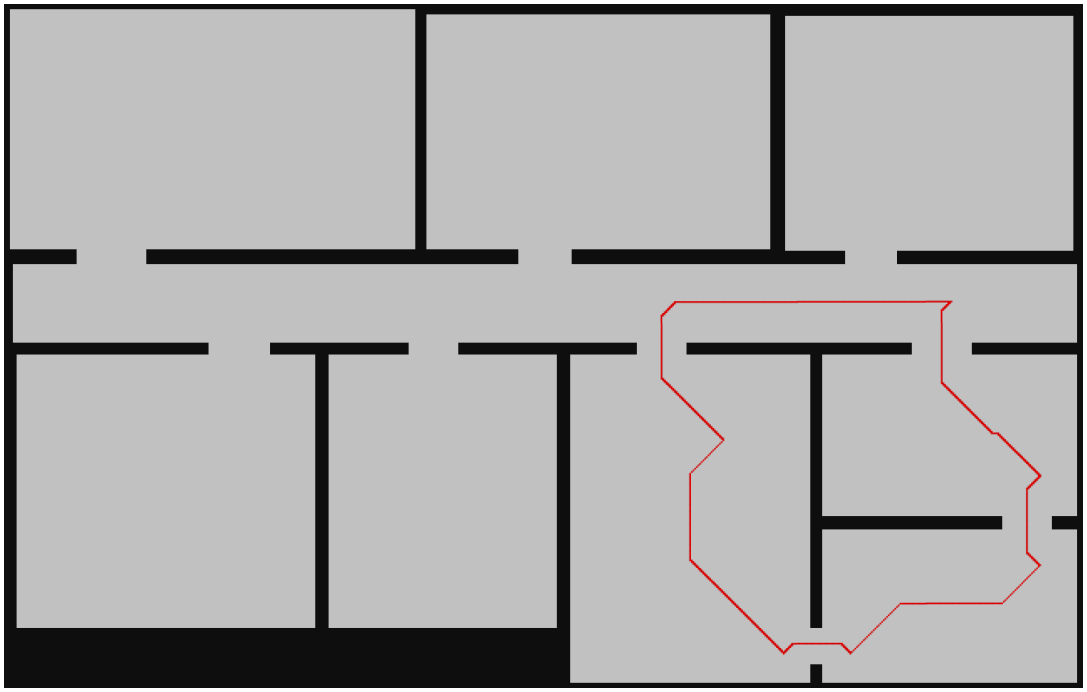
6.1 Porovnanie diagramov

Implementovaný balíček pre plánovanie cesty pomocou Voronoiových diagramov poskytuje pre ich vytvorenie dva prístupy. Prvý z nich vychádza z algoritmu od Bräunl, ktorý je popísaný v kapitole 4.1.1. Tento prístup je vhodný pre plánovanie cesty v rámci jednej miestnosti, ale v priestoroch kde sú prechody medzi jednotlivými miestnosťami, prípadne v úzkych dlhých miestnostiach zlyháva. Problém je v tom že algoritmus je založený na klasifikácii prekážok podľa zoskupených pixelov v mape a tie sú v prípade stien miestností často všetky prepojené. Ukážka prostredia, v ktorom spomínaný prístup funguje bez problémov je na obrázku 6.1. Naopak, prostredie s prechodmi a chodbami, kde táto metóda zlyháva je na obrázku 6.2.

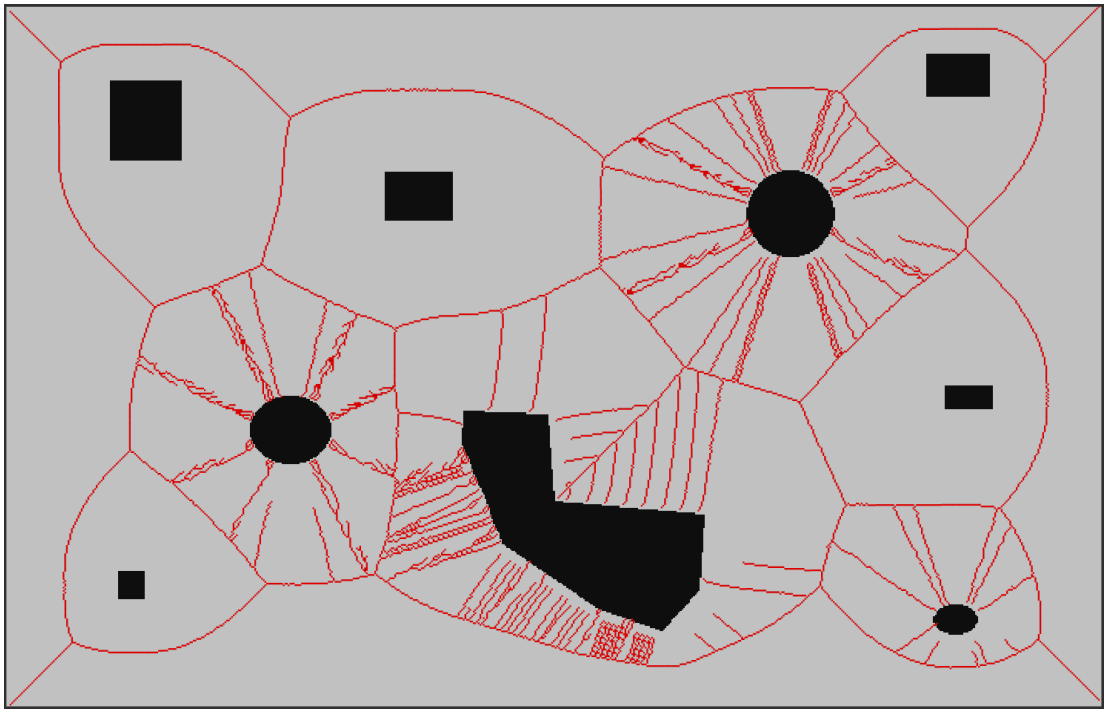
Druhý prístup, vychádzajúci z algoritmu od Lau, ktorý je predstavený v kapitole 4.1.2, funguje bez problémov ak sa Voronoiov diagram počíta pre celý areál, celú budovu, a najlepšie výsledky dosahuje vtedy, ak sú všetky objekty štvoruholníkového tvaru. Táto varianta dokáže vypočítať diagram aj pre oblasť jednej miestnosti, avšak výsledok nie je tak dobrý ako v prístupe od Bräunl. V prípade že prekážky nie sú pravouhlé štvoruholníky, tak vznikajú hrany diagramu, ktoré v skutočnosti hranami nie sú, čo je zobrazené na obrázku 6.3. Ukážka vypočítaného diagramu pre druhú mapu, zhodnú s mapou z predchádzajúceho odseku je na obrázku 6.4.



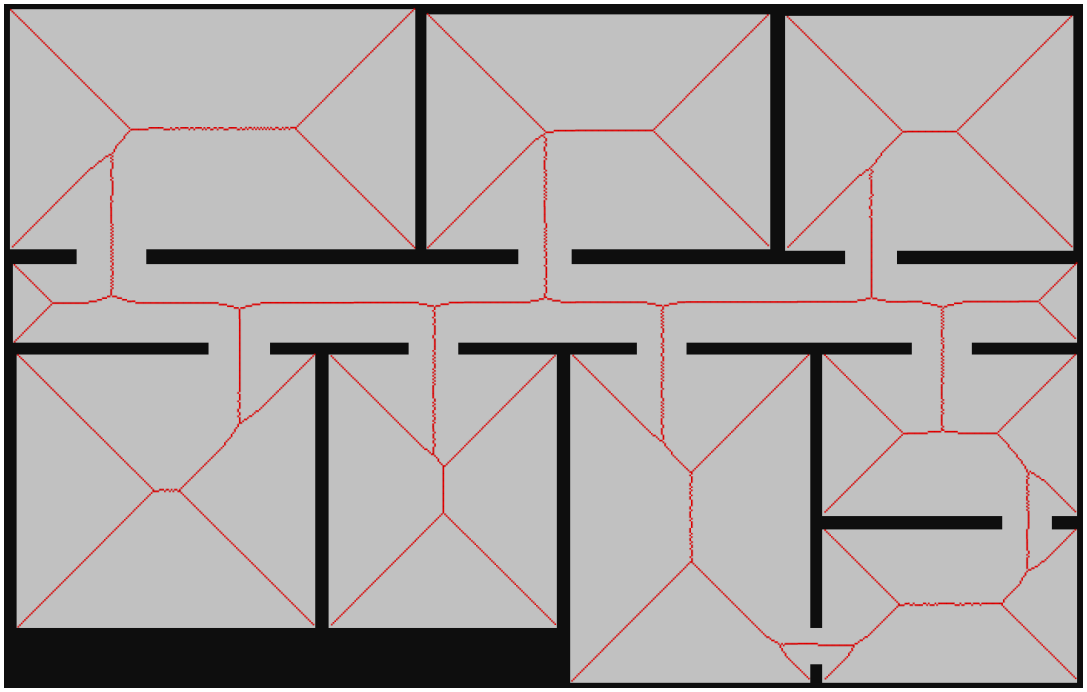
Obr. 6.1: Voronoiov diagram vytvorený prístupom Bräunl, v prostredí, kde funguje bez problémov



Obr. 6.2: Diagram vytvorený prístupom Bräunl, v prostredí, kde zlyháva



Obr. 6.3: Diagram vytvorený prístupom Lau, s nedostatkami



Obr. 6.4: Diagram vytvorený prístupom Lau, v prostredí, kde funguje bezchybne

6.2 Porovnanie plánovania

V tejto časti porovnáme plánovanie cesty pomocou troch plánovačov. Prvým z nich bude náš globálny plánovač `voronoi_path_planner`, ďalším bude plánovač `global_planner`,

s použitím A* algoritmu a tretím bude plánovač `navfn`, ktorý používa Dijkstrov algoritmus. Každý z nich bude použitý v rovnakej mape so zhodným štartovým a cieľovým bodom. Medzi veličiny ktoré budeme pozorovať patrí čas hľadania cesty od štartu k cieľu, jej dĺžka a vizuálne si porovnáme aj vzdialenosť vypočítanej cesty od prekážok.

Prvou mapou, v ktorej sa bude hľadať cesta je mapa číslo 1. Je to jednoduchá mapa jednej miestnosti s niekoľkými prekážkami, preto je pre hľadanie cesty našim plánovačom zvolené varianta Bräunl. Vytvorenie diagramu, ktoré prebieha v inicializačnej časti trvá 1,510 sekundy. Následné vyhľadanie cesty zo štartu do cieľa trvalo 0.097 sekundy, pričom cesta bola dlhá 4,12 metrov. Hľadanie cesty globálnym plánovačom pomocou A* algoritmu trvalo len 0.02 sekundy a cesta bola kratšia o 0,37m. Čas potrebný pre plánovač používajúci Dijkstrov algoritmus bol 0,058 sekundy a výsledná cesta bola ešte o 0,2 metra kratšia ako tá predchádzajúca. Vypočítané cesty sú vyobrazené na obrázkoch v prílohe A.

Mapa č.2 ktorá bola využitá pre testovanie predstavuje pôdorys budovy s miestnosťami a úzkou chodbou. Z tohto dôvodu bola pre náš plánovač zvolená varianta Lau. Vytvorenie Voronoiovho diagramu v tejto mape trvalo približne rovnako dlho ako v predchádzajúcej mape, teda 1,514 sekundy a nájdenie cesty s využitím tohto diagramu zabralo 0,085 sekundy. Výsledná cesta má 49,35m a vedie stredom medzi dvoma prekážkami, čo možno vidieť na obrázku A.4. Nájdenie cesty pomocou globálneho plánovača s A* algoritmom trvalo o približne 0,05 sekundy kratšie a cesta má takmer o 12 metrov menej. Hoci nájdenie cesty trvalo kratšie a aj výsledná cesta je kratšia, vedie v niektorých miestach blízko popri stenách, čo je možné vidieť na obrázku A.5. Podobnú cestu dlhú 37,04m našiel aj globálny plánovač *navfn* s Dijkstrovým algoritmom a trvalo mu to 0,053 sekundy.

Výsledky plánovania v mape č.3 a mape č.4, ako aj všetky ostatné výsledky je možné vidieť v tabuľke 6.1. V prípade mapy číslo 5 a 6, nedokázal globálny plánovač pomocou algoritmu A* nájsť vhodnú cestu zo štartu k cieľu, preto je v tabuľke namiesto času a vzdialenosti zapísané X a taktiež pre tieto varianty nie je vykreslená mapa v prílohách.

Mapa	Vytáranie diagramu	Voronoi		global planner - A*		navfn - Dijkstra	
		čas	dĺžka	čas	dĺžka	čas	dĺžka
1.	1.510	0.097	4.12	0.020	3.75	0.058	3.55
2.	1.514	0.085	49.35	0.037	37.91	0.053	37.04
3.	3.108	0.132	60.76	0.122	54.76	0.157	52.57
4.	1.539	0.244	51.82	X	X	0.105	45.85
5.	5.263	0.538	145.63	X	X	0.290	125.68

Tabuľka 6.1: Výsledky porovnávania plánovačov

Z dát zaznamenaných v tabuľke je možné zhodnotiť, že cesta ktorá vedie po hranách Voronoiovho diagramu je o 10 až 20 percent dlhšia ako tá vypočítaná pomocou algoritmu A*, alebo Dijkstrovho algoritmu. Avšak po posúdení výslednej cesty podľa obrázkov v prílohách je zrejmé, že táto cesta je pre pohyb robota oveľa bezpečnejšia, keďže vedie v maximálnej vzdialenosti od prekážok, zatiaľ čo cesty od ostatných plánovačov sú vzdialené od prekážok podľa toho ako je nastavená hodnota parametra `inflation_radius`.

Kapitola 7

Záver

Plánovanie cesty nie je elementárnou úlohou v procese navigácie robota k cieľu, a v jeho priebehu môže dôjsť k rôznym úskaliam. Častým problémom je, že robot sa dostane do kolízie s prekážkou. Pre minimalizovanie tohto nedostatku bol v rámci tejto bakalárskej práce implementovaný zásuvný modul pre robotický operačný systém, ktorý pre nájdenie cesty k cieľu využíva hrany Voronoiového diagramu. Tieto hrany vedú stredom medzi dvoma prekážkami, čo minimalizuje riziko zrážky. Modul bol testovaný pomocou simulátora Gazebo pre ROS, a naplánovaná cesta bola zobrazovaná vo aplikácii pre vizualizáciu dát RViz.

Z výsledkov testovania je možné zhodnotiť že cesta vypočítaná pomocou implementovaného modulu s využitím Voronoiových diagramov je dlhšia ako tá vypočítaná pomocou algoritmu A*, alebo Dijkstrovho algoritmu, avšak vedie v maximálnej vzdialenosti od prekážok čo minimalizuje riziko kolízie, zatiaľ čo cesty od ostatných plánovačov sú vzdialené od prekážok podľa toho ako je nastavená hodnota parametra `inflation_radius`. Pri použití toho plánovača je možné tento parameter, ktorý udáva veľkosť „nafúknutia“ prekážok, aby nedochádzalo ku zrážkam, vynulovať.

Do budúcna by ešte bolo vhodné vylepšiť v tomto module, hľadanie cesty z miesta ktoré neleží na Voronoiovom diagrame na jeho hranu, čo je aktuálne implementované pomocou Dijkstrovho algoritmu a v niektorých prípadoch to generuje nevýhodnú cestu, kedy sa robot vzdiaľuje od cieľa.

Literatúra

[1] *Catkin: Conceptual overview*. [Online; navštívené 23.11.2017].
URL http://wiki.ros.org/catkin/conceptual_overview

[2] *Catkin: Workspaces*. [Online; navštívené 23.11.2017].
URL http://wiki.ros.org/catkin/workspaces

[3] *Changes between ROS 1 and ROS 2*. [Online; navštívené 02.03.2018].
URL http://design.ros2.org/articles/changes.html

[4] *Creating a ROS Package*. [Online; navštívené 21.11.2017].
URL http://wiki.ros.org/ROS/Tutorials/CreatingPackage

[5] *Releases*. [Online; navštívené 02.03.2018]. Aktualizované 10.12.2017.
URL https://github.com/ros2/ros2/wiki/Releases

[6] *ROS Answers*. [Online; navštívené 19.11.2017].
URL https://answers.ros.org/users/

[7] *ros_base: pluginlib*. [Online; navštívené 28.12.2017].
URL http://wiki.ros.org/pluginlib

[8] *ROS Filesystem Concepts: Manifest*. [Online; navštívené 23.11.2017].
URL http://wiki.ros.org/Manifest

[9] *ROS Filesystem Concepts: msg*. [Online; navštívené 25.11.2017].
URL http://wiki.ros.org/srv

[10] *ROS Filesystem Concepts: Packages*. [Online; navštívené 21.11.2017].
URL http://wiki.ros.org/Packages

[11] *ROS Graph Concepts: Master*. [Online; navštívené 27.11.2017].
URL http://wiki.ros.org/Master

[12] *ROS Graph Concepts: Nodes*. [Online; navštívené 27.11.2017].
URL http://wiki.ros.org/Nodes

[13] *ROS Graph Concepts: Parameter server*. [Online; navštívené 27.11.2017].
URL http://wiki.ros.org/Parameter%20Server

[14] *ROS Graph Concepts: Services*. [Online; navštívené 27.11.2017].
URL http://wiki.ros.org/Services

[15] *ROS Graph Concepts: Topics*. [Online; navštívené 27.11.2017].
URL http://wiki.ros.org/Topics

[16] *ROS Users of the World*. [Online; navštívené 19.11.2017].
URL http://metrorobots.com/rosmap.html

[17] *ROS Wiki: base_local_planner*. [Online; navštívené 28.12.2017]. Aktualizované 25.02.2017.
URL http://wiki.ros.org/base_local_planner

[18] *ROS wiki: Concepts*. [Online; navštívené 19.11.2017].
URL http://wiki.ros.org/ROS/Concepts

[19] *ROS Wiki: Distributions*. [Online; navštívené 19.11.2017].
URL http://wiki.ros.org/Distributions

[20] *ROS Wiki: move_base*. [Online; navštívené 28.12.2017]. Aktualizované 13.05.2016.
URL http://wiki.ros.org/move_base

[21] *ROS Wiki: nav_core*. [Online; navštívené 28.12.2017]. Aktualizované 13.05.2016.
URL http://wiki.ros.org/nav_core

[22] *ROS Wiki: navigation*. [Online; navštívené 28.12.2017].
URL http://wiki.ros.org/navigation

[23] de Berg, M.: *Computational geometry: algorithms and applications*. Berlin: Springer, tretie vydanie, 2008, ISBN 978-3-642-09681-5.

[24] Bräunl, T.: *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. Springer, 2008, ISBN 978-3-540-70533-8.

[25] Choset, H.: *Principles of robot motion : theory, algorithms and implementations*. MIT Press, 2005, ISBN 0-262-03327-5.

[26] Kalra, N.; Ferguson, D.; Stentz, A.: Incremental reconstruction of generalized Voronoi diagrams on grids. ročník 57, 2006: s. 123–128.
URL https://www.sciencedirect.com/science/article/pii/S0921889007000966

[27] Lau, B.; Sprunk, C.; Burgard, W.: *Efficient grid-based spatial representations for robot navigation in dynamic environments*. *Robotics and Autonomous Systems*, ročník 61, č. 10, 2013: s. 1116 – 1130, ISSN 0921-8890.
URL http://www.sciencedirect.com/science/article/pii/S092188901200142X

[28] Okabe, A.: *Spatial tessellations : concepts and applications of Voronoi diagrams*. Wiley, druhé vydanie, 2000, ISBN 0-471-98635-6.

[29] Thomas, D.: *ROS: Introduction*. 2014, [Online; navštívené 19.11.2017].
URL http://wiki.ros.org/ROS/Introduction

[30] Zhang, T. Y.; Suen, C. Y.: A Fast Parallel Algorithm for Thinning Digital Patterns. *Commun. ACM*, ročník 27, č. 3, 1984: s. 236–239, ISSN 0001-0782.
URL https://dl.acm.org/citation.cfm?id=358023

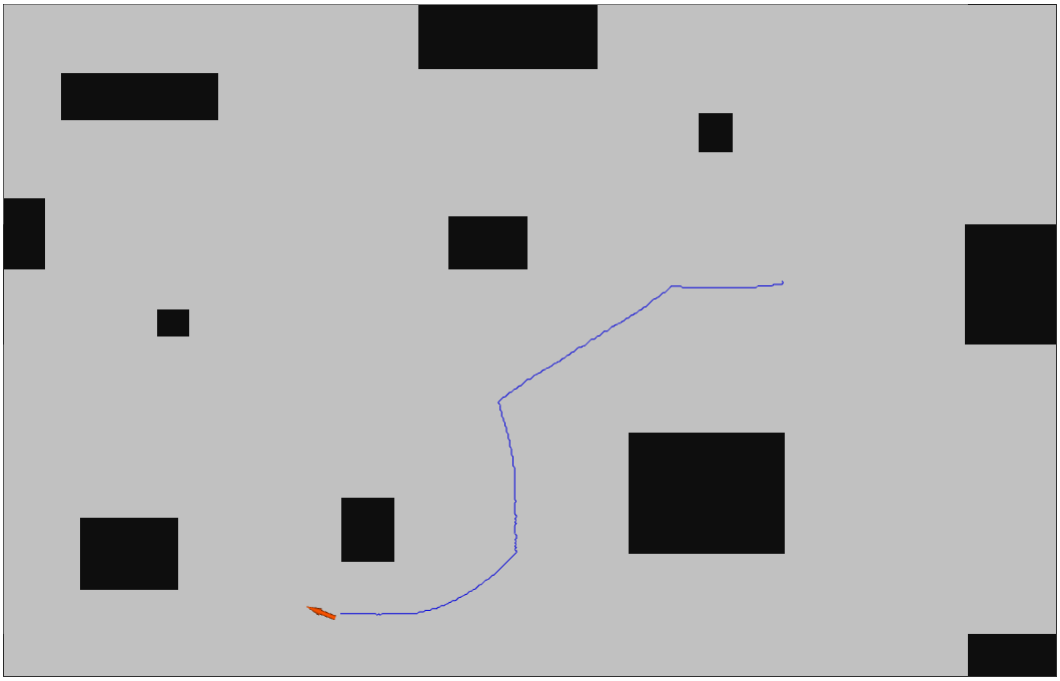
Prílohy

Zoznam príloh

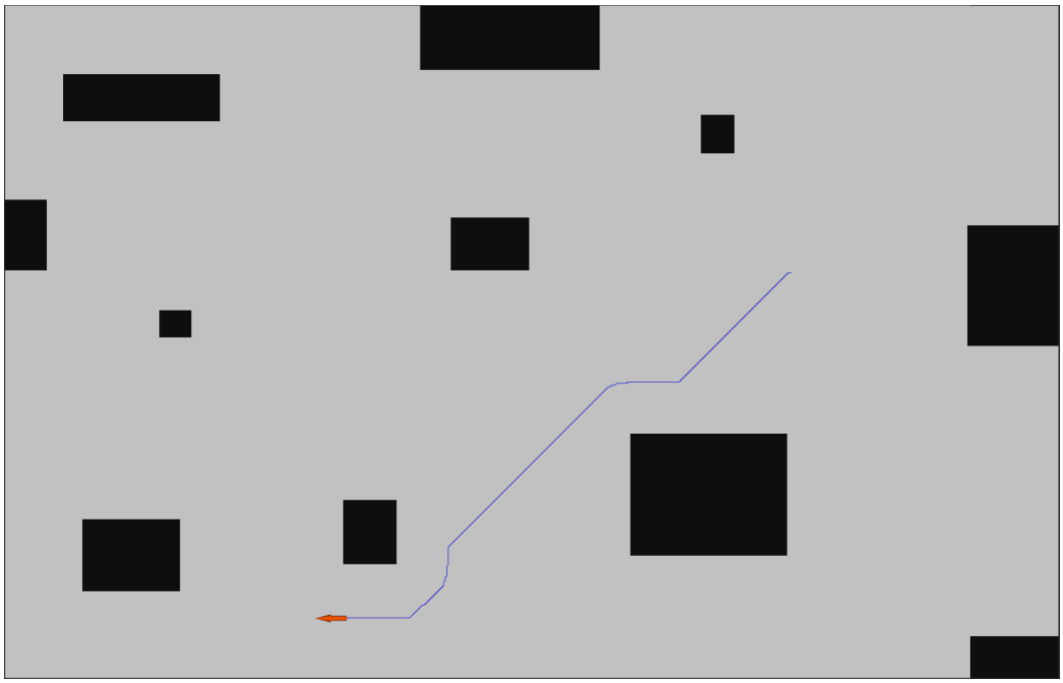
Príloha A

Zobrazenie vypočítaných ciest

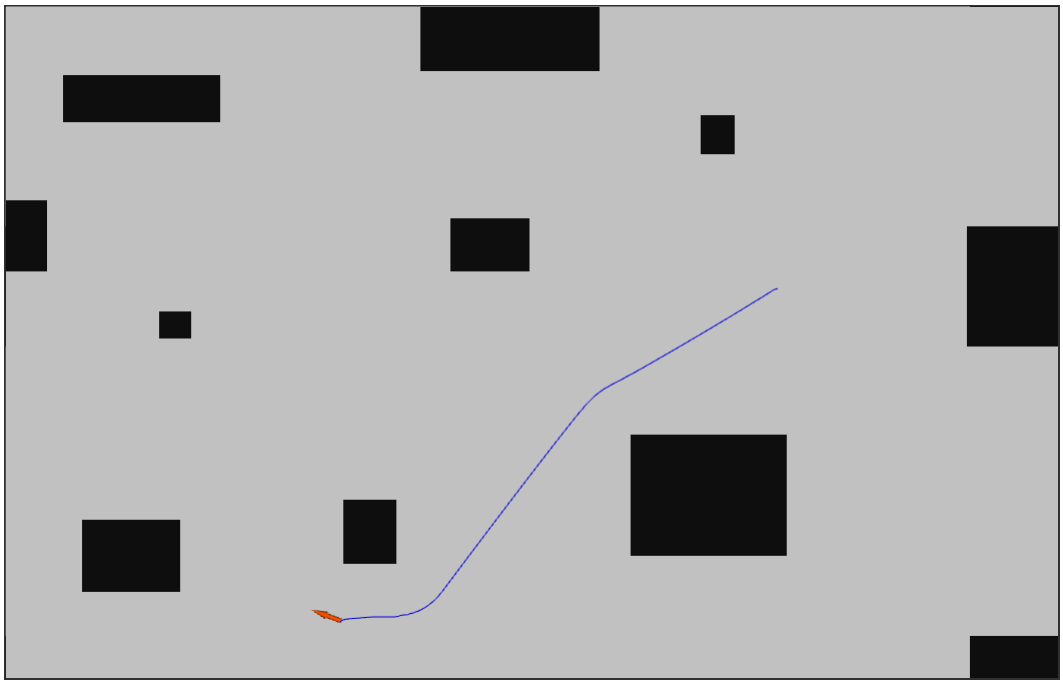
Cesty v mape číslo 1



Obr. A.1: Cesta nájdená s využitím Voronoiovho diagramu vypočítaného prístupom Bräunl

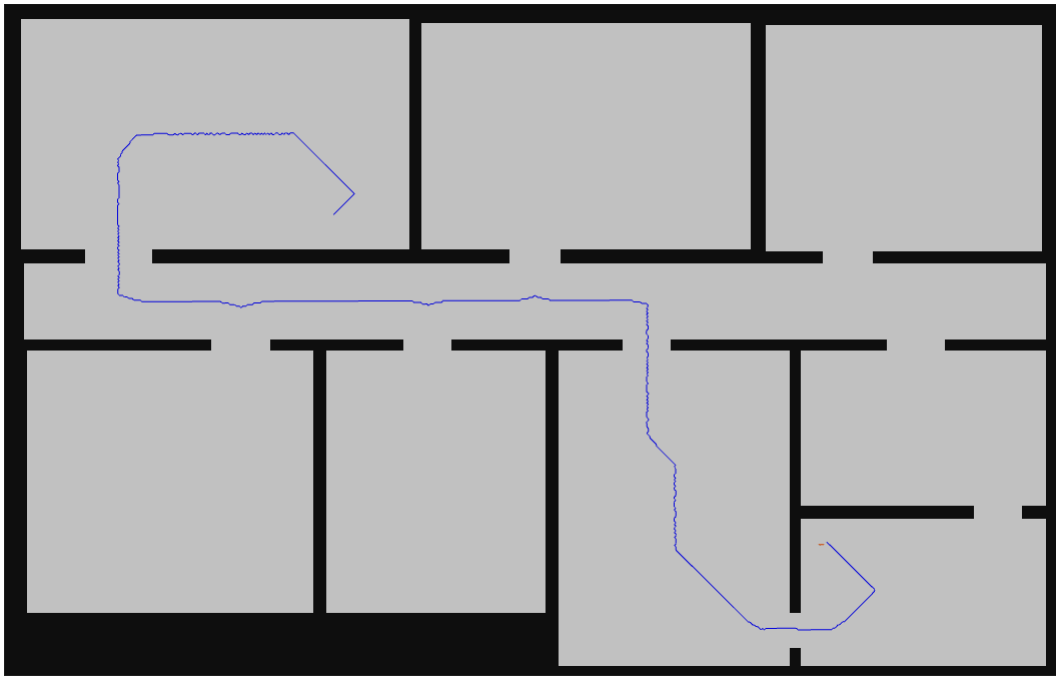


Obr. A.2: Cesta vypočítaná metódou A*

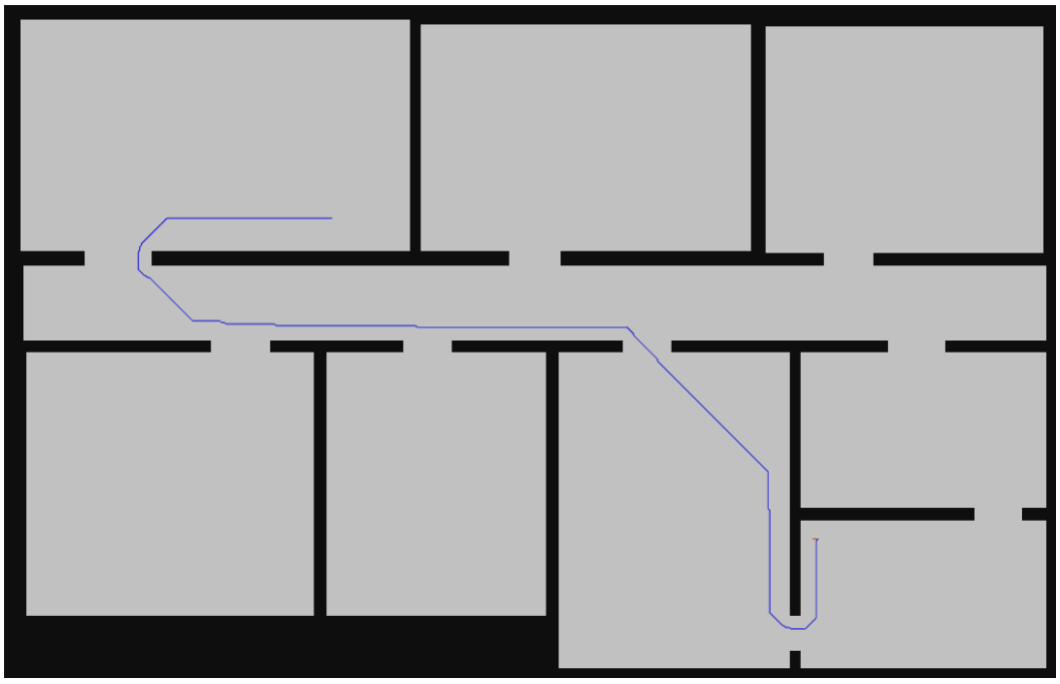


Obr. A.3: Cesta vypočítaná Dijkstrovou metódou

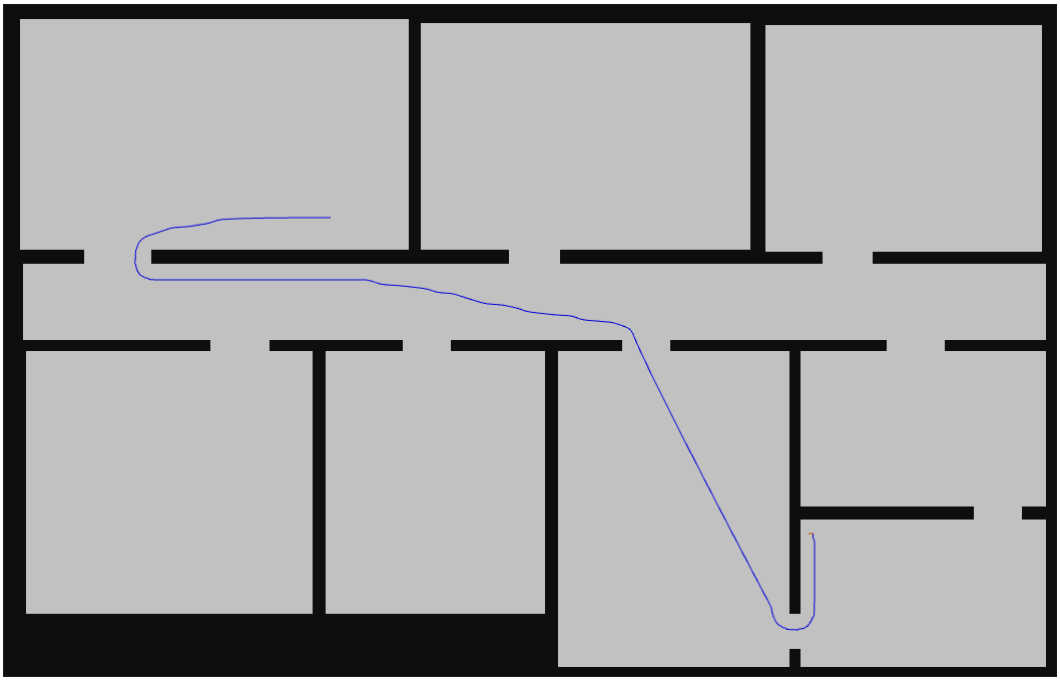
Cesty v mape číslo 2



Obr. A.4: Cesta nájdená s využitím Voronoiovho diagramu vypočítaného prístupom Lau

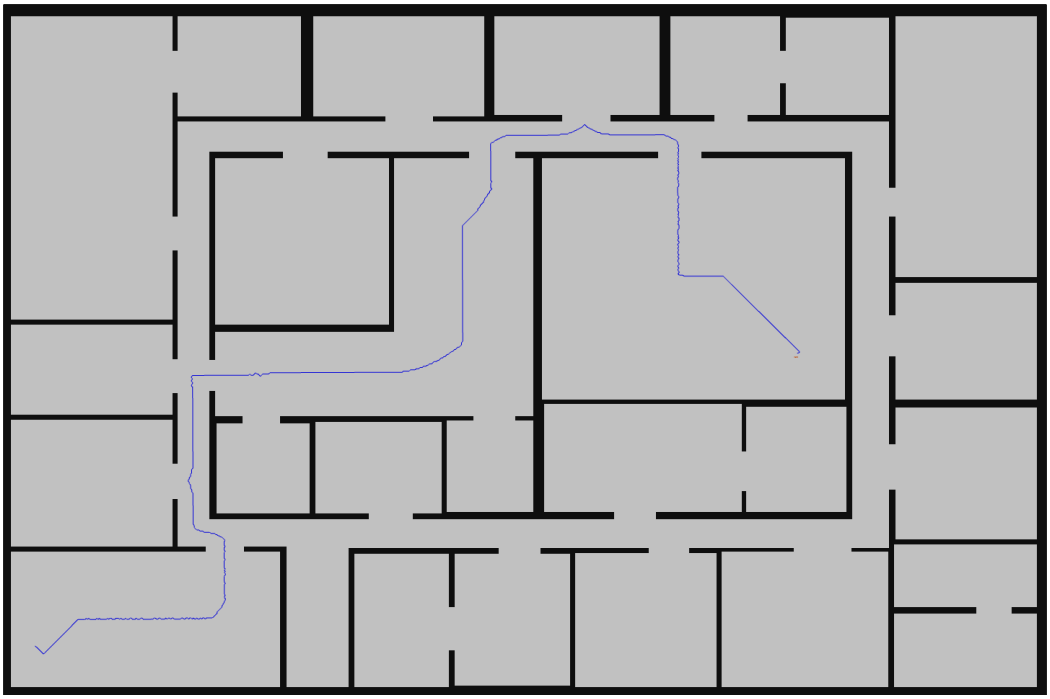


Obr. A.5: Cesta vypočítaná metódou A*

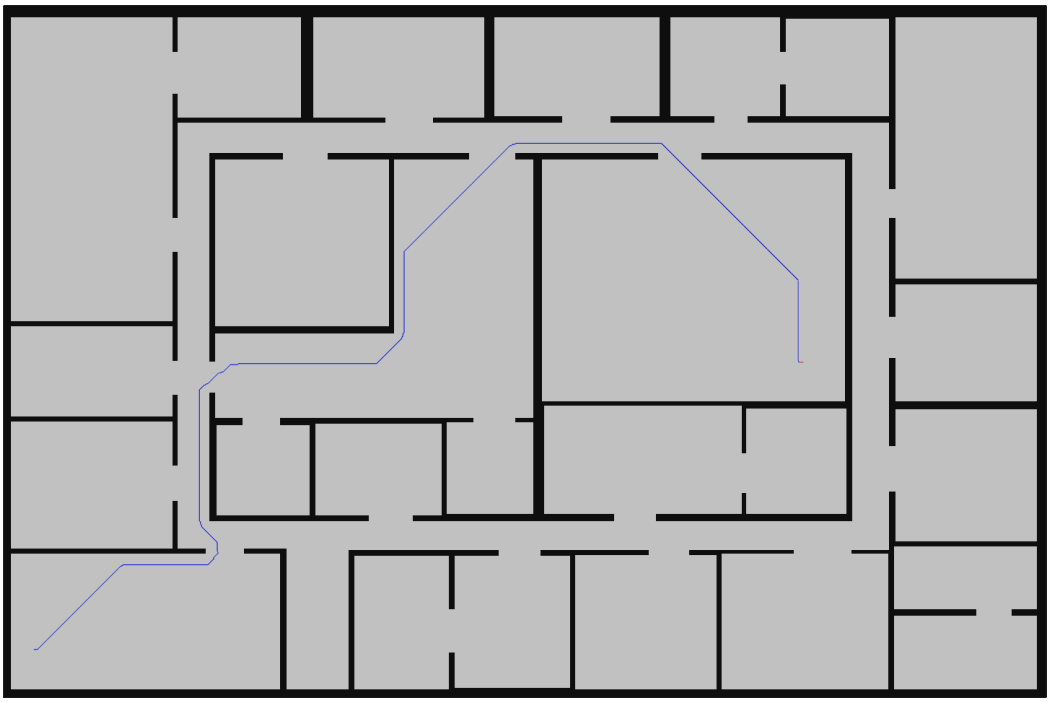


Obr. A.6: Cesta vypočítaná Dijkstrovou metódou

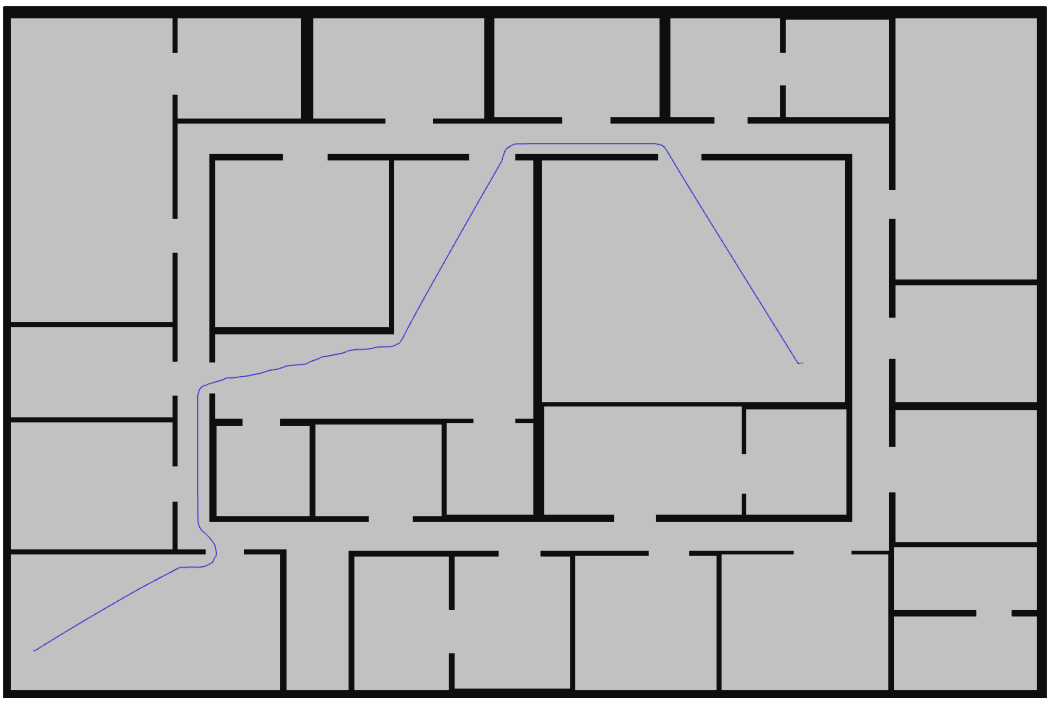
Cesty v mape číslo 3



Obr. A.7: Cesta nájdená s využitím Voronoiovho diagramu vypočítaného prístupom Lau

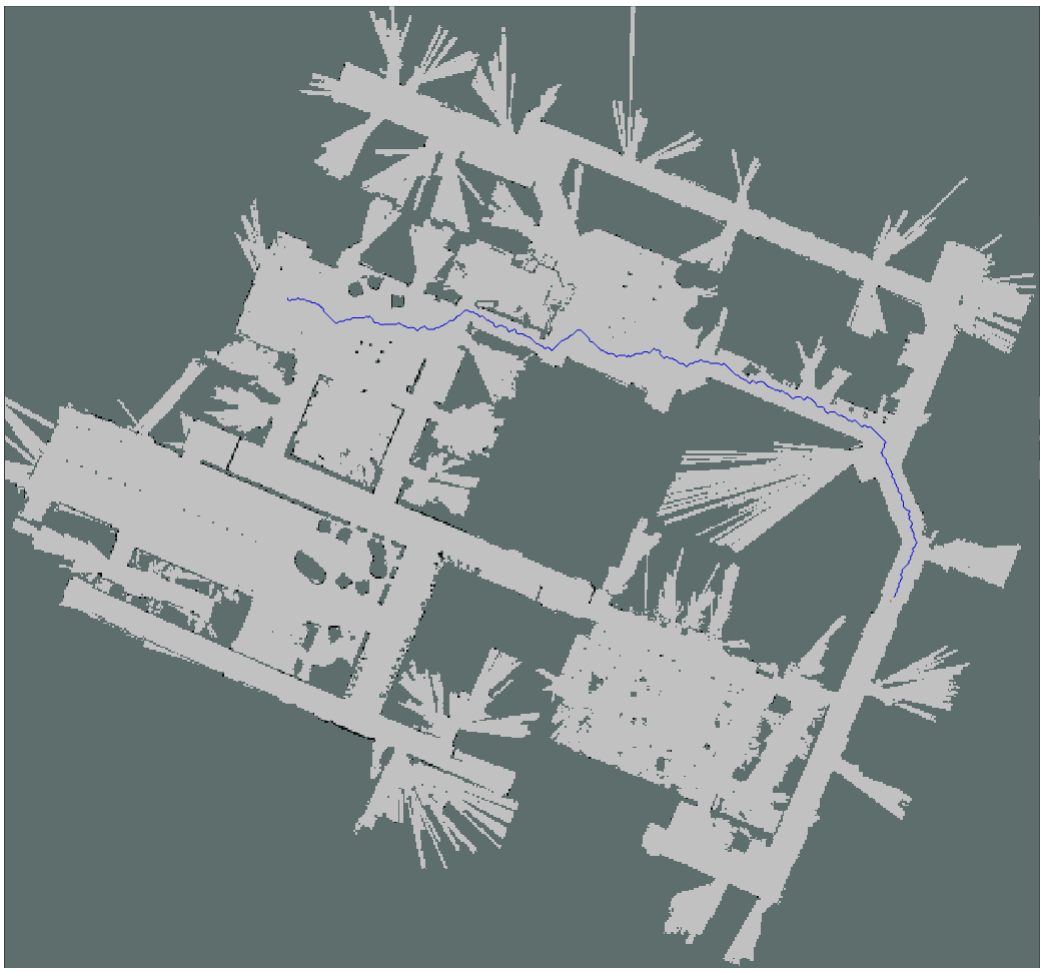


Obr. A.8: Cesta vypočítaná metódou A*

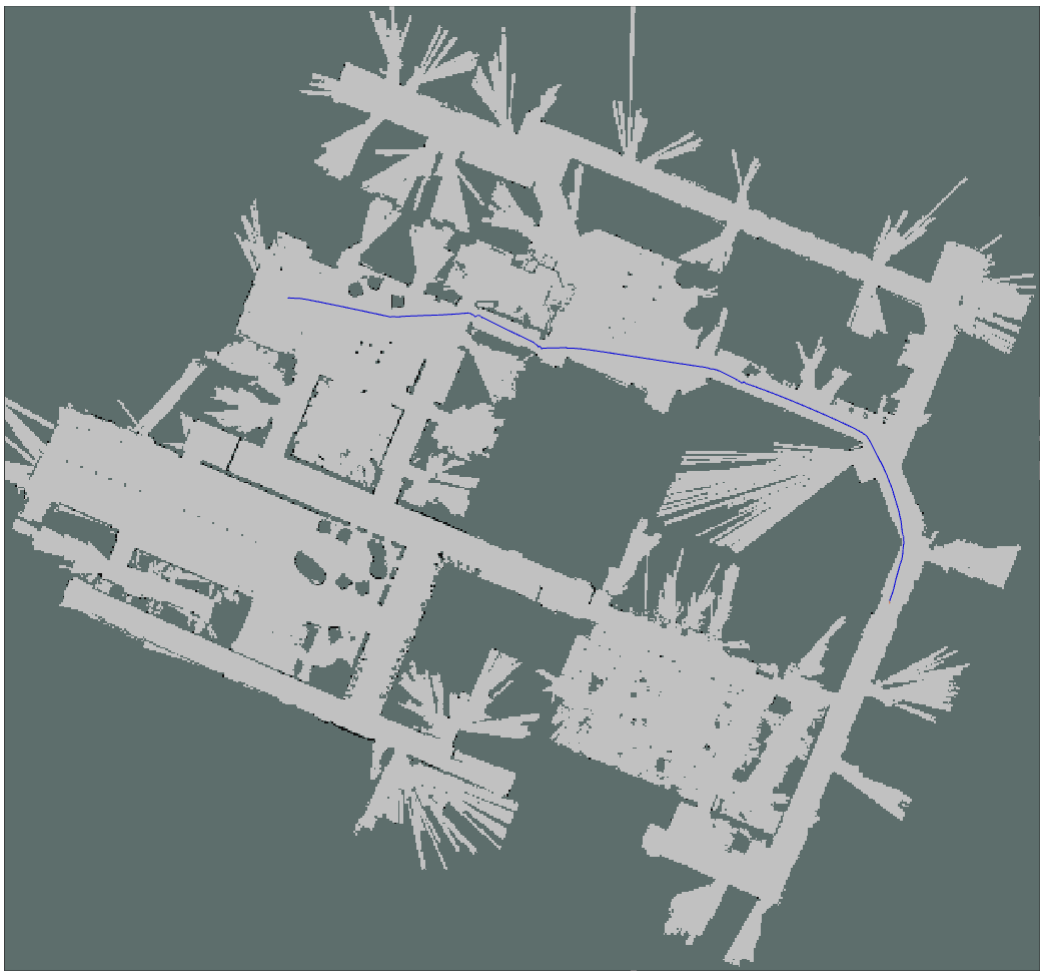


Obr. A.9: Cesta vypočítaná Dijkstrovou metódou

Cesty v mape číslo 4

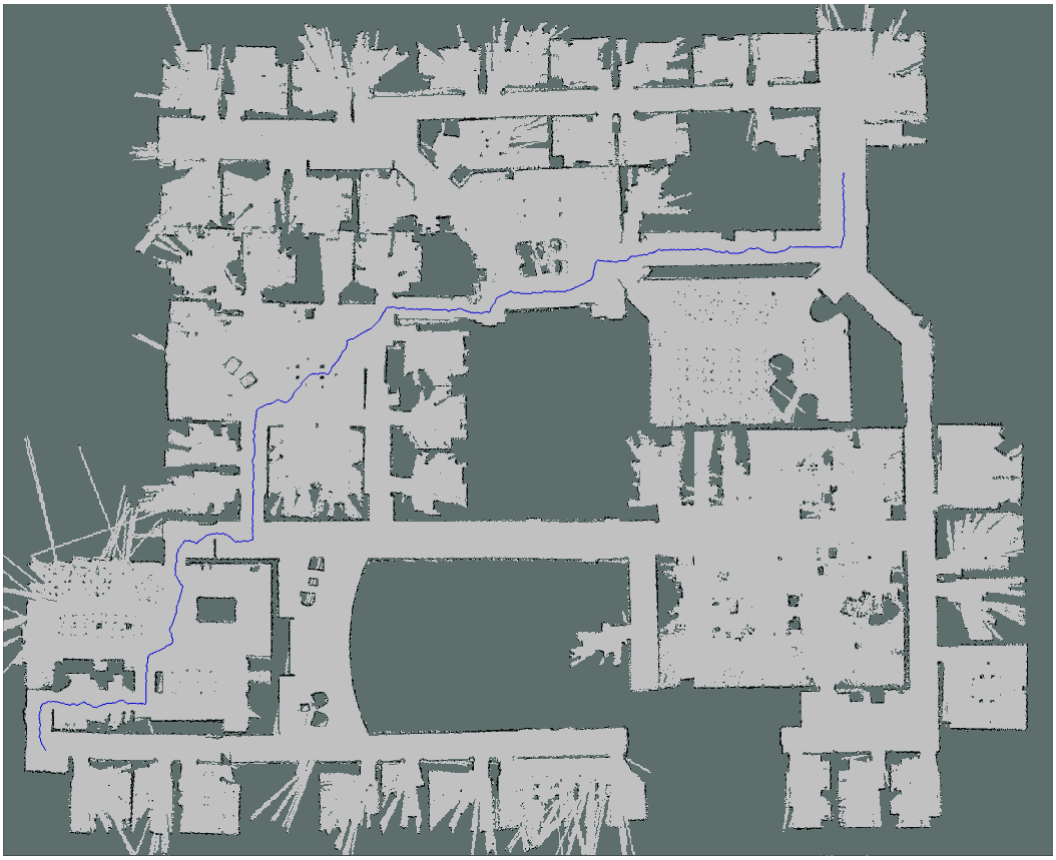


Obr. A.10: Cesta nájdená s využitím Voronoiovho diagramu vypočítaného prístupom Lau

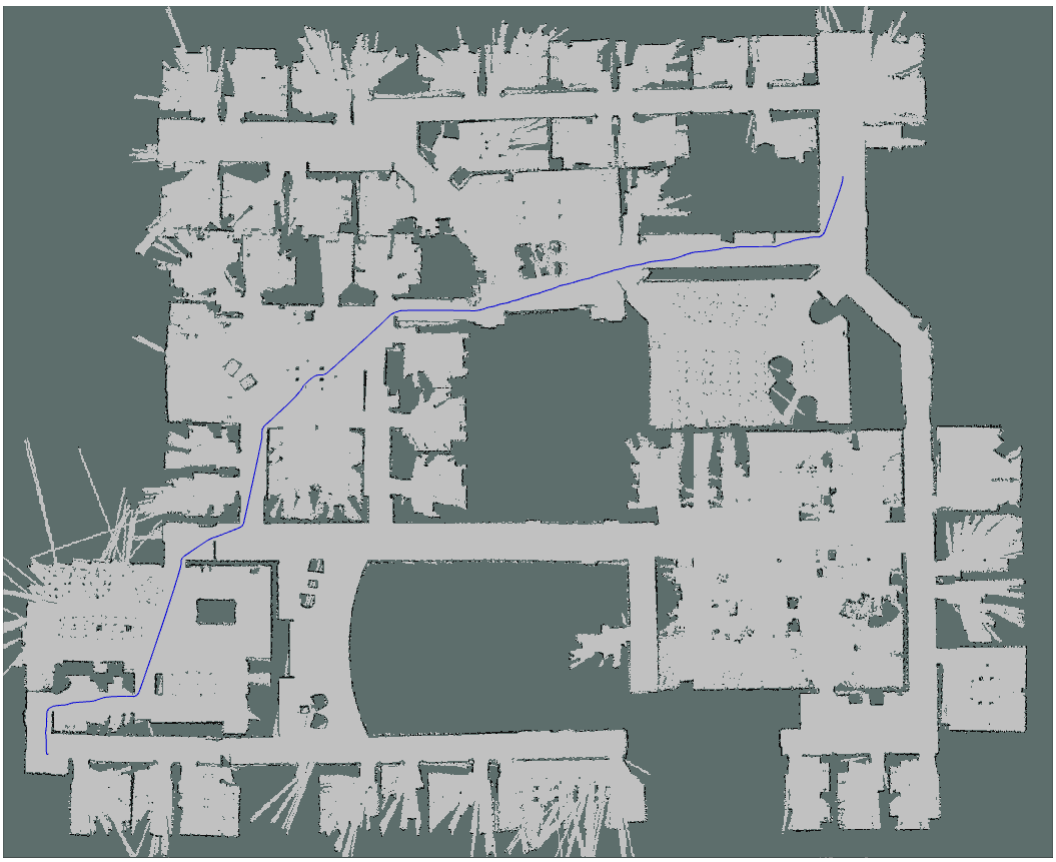


Obr. A.11: Cesta vypočítaná Dijkstrovou metódou

Cesty v mape číslo 5



Obr. A.12: Cesta nájdená s využitím Voronoiovho diagramu vypočítaného prístupom Lau



Obr. A.13: Cesta vypočítaná Dijkstrovou metódou

Príloha B

Návod na použitie zásuvného modulu

V tejto časti si popíšeme úkony, ktoré je potrebné vykonať, aby sme pre plánovanie cesty mohli použiť implementovaný plánovač využívajúci Voronoiové diagramy. Názvy súborov, v ktorých sa vykonávajú zmeny v tejto kapitole sú pre použitie robota Turtlebot, takže pri použití inej platformy, sa názvy a umiestenie súborov môže líšiť, avšak zmeny ktoré sa v súboroch vykonávajú ostávajú nezmenené.

Stiahnutie

Stiahnuté súbory balíčka, konkrétne priečinok `voronoi_path_planing/` so všetkými súbormi, je potrebné umiestniť do pracovného prostredia (*workspace*), v našom prípade to bude `catkin_ws`.

Preklad

Preklad zdrojových súborov sa spúšťa príkazom `catkin_make` v priečinku `catkin_ws`.

Nastavenie parametrov

U robota Turtlebot sa pomocou následujúcich príkazov dostaneme do priečinka kde je umiestnený konfiguračný súbor `move_base.launch.xml` pre navigáciu.

```
$ roscd turtlebot_navigation/
$ cd launch/includes/
```

V súbore `move_base.launch.xml` je potrebné upraviť parameter s názvom `"base_global_planner"`, prípadne ak sa tam nenachádza, tak je treba pridať následujúci riadok, ktorý umožní použitie nášho globálneho plánovača, namiesto implicitnej varianty.

```
<param name="base_global_planner"
value="voronoi_path_planner/VoronoiPathPlanner"/>
```

Taktiež je možné pridať následujúci parameter, ktorý umožní zmeniť variantu algoritmu v našom plánovači z predvolenej Lau, na Bräunl.

```
<rosparam
file="$(find voronoi_path_planing)/param/voronoi_path_planning_params.yaml"
command="load"/>
```

Pre zmenu varianty z Lau na Bräunl je potom potrebné do súboru `move_base.launch.xml` pridať ešte tento parameter.

```
<param name="voronoi_planner_method" value="Braunl"/>
```

Posledným parametrom, ktorý je vhodné pridať do konfiguračného súboru je parameter `"inflation_radius"`.

```
<param name="global_costmap/inflation_layer/inflation_radius" value="0.0"/>
```

To sú všetky zmeny ktoré je potrebné vykonať, aby plánovanie cesty prebiehalo využitím Voronoiových diagramov.

Zobrazovanie v RViz

Pre zobrazenie Voronoiového diagramu a vypočítanej cesty v aplikácii RViz, je potrebné pridať novú položku, do zoznamu zobrazovaných dát. Postup je následovný:

- Klikneme na tlačidlo *Add* v ľavom dolnom rohu okna.
- Zo zoznamu v novom okne vyberieme, aký druh dát chceme zobrazovať.

  Pre vykreslenie Voronoiového diagramu zvolíme typ *Marker*.

  Pre vykreslenie vypočítanej cesty zvolíme typ *Path*.
- Novo pridanú položku môžeme premenovať v spodnej časti okna.
- Novej položke nastavíme *topic*, ktorej dáta budú vykresľované.

  Pre zobrazenie bodov Voronoiového diagramu vyberáme možnosť `/VoronoiPathPlanner/Voronoi_diagram`

  Pre zobrazenie vypočítanej cesty zvolíme `/VoronoiPathPlanner/Voronoi_full_path`

Príloha C

Obsah priloženého pamäťového média

K tejto práci je priložené CD s nasledujúcim obsahom:

- `/voronoi_path_planing/` - priečinok so zdrojovými súbormi vytvoreného balíčka pre robotický operačný systém
- `/latex/` - priečinok so zdrojovými súbormi textovej časti bakalárskej práce
- `/doc/` - priečinok s dokumentáciou vytvorenou pomocou nástroja Doxygen
- `/xzivca03.pdf` - súbor s textom bakalárskej práce
- `/README.md` - súbor so stručným popisom vytvoreného balíčka a návodom na jeho použitie